ANIMATING THE EVOLUTION OF SOFTWARE

PhD Thesis

Christopher Taylor

Department of Computer Science
University of Durham
Durham, DH1 3LE, UK

1999 – 2003

Abstract

The use and development of open source software has increased significantly in the last decade. The high frequency of changes and releases across a distributed environment requires good project management tools in order to control the process adequately. However, even with these tools in place, the nature of the development and the fact that developers will often work on many other projects simultaneously, means that the developers are unlikely to have a clear picture of the current state of the project at any time. Furthermore, the poor documentation associated with many projects has a detrimental effect when encouraging new developers to contribute to the software.

A typical version control repository contains a mine of information that is not always obvious, and not easy to comprehend in its raw form. However, presenting this historical data in a suitable format by using software visualisation techniques allows the evolution of the software over a number of releases to be shown. This allows the changes that have been made to the software to be identified clearly, thus ensuring that the effect of those changes will also be emphasised. This then enables both managers and developers to gain a more detailed view of the current state of the project.

The visualisation of evolving software introduces a number of new issues. This thesis investigates some of these issues in detail, and recommends a number of solutions in order to alleviate the problems that may otherwise arise. The solutions are then demonstrated in the definition of two new visualisations. These use historical data contained within version control repositories to show the evolution of the software at a number of levels of granularity. Additionally, animation is used as an integral part of both visualisations - not only to show the evolution by representing the progression of time, but also to highlight the changes that have occurred.

Previously, the use of animation within software visualisation has been primarily restricted to small-scale, hand generated visualisations. However, this thesis shows the viability of using animation within software visualisation with automated visualisations on a large scale. In addition, evaluation of the visualisations has shown that they are suitable for showing the changes that have occurred in the software over a period of time, and subsequently how the software has evolved. These visualisations are therefore suitable for use by developers and managers involved with open source software. In addition, they also provide a basis for future research in evolutionary visualisations, software evolution and open source development.

Acknowledgements

This thesis has been the result of many months of work. However, without the support of many others it is unlikely that it would ever have been completed. My thanks therefore go to the following people, and many others, whose help has been invaluable during this period of study.

Firstly, my thanks go to my supervisor Malcolm Munro for all his help, direction and advice. His enthusiasm led me to study for a PhD in the field of software visualisation and his knowledge and understanding of this research area was invaluable. His willingness to make time to discuss concepts and read through many chapters, despite his numerous other activities, was crucial in completing this thesis. Thank you to Jill Munro also, who had the 'privilege' of proof reading parts of this work!

Thanks also go to both Brendan Hodgson and Sarah Drummond, who were responsible for managing my teaching duties during my time at the department. They were both very supportive, and allowed me to balance my research and teaching duties as unobtrusively as possible.

I'd also like to thank Andrew Hatch, Claire Knight, James Witter, Mike Smith and everyone else at the Visualisation Research Group at the Computer Science department in Durham. Their feedback, support and suggestions proved vital in refining many areas of this thesis, whilst their friendship provided times of light relief on many an occasion. In addition, the opportunity to get out of Durham on several occasions in order to discuss many of these ideas in a more relaxed environment was most valuable, and thank you to Claire for organising many of these events.

In addition, thanks go to my parents for developing my early interests in computer science, and supporting me wholeheartedly throughout my time in academia. They have been a constant source of encouragement, and their advice has been much appreciated.

Finally, I'd like to thank my wife, Abi, for all her help and assistance during this research. In addition to proof reading this thesis with great diligence, she has put up with the times of frustration and despair when the research was not going as well as it might have been. Her understanding and support at these times was very much appreciated! My friends during my time of study, including Rich, William and Laura have also had much to put up with at various times, although they are still continuing to talk to me!

Copyright

Declaration

No part of the material provided has previously been submitted by the author for a higher degree in the University of Durham or in any other University. All the work presented here is the sole work of the author and no one else.

This research has been documented, in part, within the following publications:

- A. Hatch, M. Smith, C. Taylor, M. Munro, *'No Silver Bullet for Software Visualisation Evaluation'*, Proceedings of The International Conference on Imaging Science, Systems, and Technology (CISST), Las Vegas, USA, June 2001
- C. Taylor, M. Munro, '*Revision Towers*', Proceedings of the IEEE 1st International Workshop on Visualizing Software for Understanding and Analysis, Paris, France, June 2002

Table of Contents

List of Figures

List of Tables

ANIMATING THE EVOLUTION OF SOFTWARE

1. Introduction

1.1. Background

The continual growth of online communication has allowed new methods of software development to form. Open source software development is an example of this, which allows every user of the software full access to the source code. Rather than development being restricted to a limited number of maintainers, this instead allows development to take place on a much larger scale. Users from anywhere within the world are able to examine the code and supply complete modifications, allowing them to influence the reliability and functionality of the software.

An open source development model raises a number of issues. Open source developers are predominantly volunteers, and are often involved in a wide range of projects at any time. Additionally, it is not possible to force developers to work on any particular part of the software. The process is therefore unlike traditional software development, where change requests are formed and assigned to a developer responsible for the work. Instead, within open source development, the users are also the developers. If a user is unhappy with the performance of a particular feature, they are in the position to modify that feature themselves without prior authorisation, and submit their code for inclusion into the project.

In order to modify software successfully, a developer must be able to comprehend the behaviour of the existing software. However, the large numbers of modifications associated with open source development means that a developer may not be fully aware of the exact behaviour of the software at any point. Therefore, the success of their modifications can not be guaranteed. Furthermore, should a developer stop contributing to the project due to other commitments, and return at a later date, they will not immediately be in a position to restart their work on the project. Instead, they will have to identify the changes made during the time that they were absent, and then integrate those changes into their original understanding of the software.

A solution to these problems is difficult with existing tools. The predominant means of change identification is through the use of text-based tools, showing changes within two versions of a file with a purely lexical line by line comparison technique. However, the field of software visualisation allows not only the textual source code, but also the relationships within the software to be identified graphically, allowing the user to gain a more comprehensive understanding of the software.

Therefore, visualising software change may provide a solution not only to the clear identification of the modifications made, but also the impact of those modifications within the software.

1.1.1. Software evolution

Although open source development presents a more extreme view of software evolution, evolution is inevitable in any development method if the software is to remain useful [Lehman00]. Furthermore, the virtual nature of software means that the extent of this evolution is potentially infinite.

Software may continue to be used long after the predicted life span, and features and behaviour that were never designed or even considered during the initial requirements will be introduced. Adding these features causes new relationships to be introduced within the software, so increasing complexity. The new features also require more code, and therefore the software increases in size, and becomes increasingly difficult to comprehend. Furthermore, changes may impact unexpected areas of the software, as previous assumptions about data values and behaviour become invalidated.

As may be expected, the problems associated with evolving software are costly to the extent that software maintenance will often consume the majority of the software budget [Pigoski97]. Therefore, a reduction in the difficulty of identifying and understanding the changes that occur within software will be beneficial.

1.1.2. Change comprehension

Program comprehension research indicates that a developer needs a good understanding of the current state of the software before it is possible to modify the software successfully. This understanding may be achieved using a number of methods, although all are costly in terms of time and mental effort. However, as soon as a change is made, the developer must then become reacquainted with the new version. If the actual change that was made was unrecorded, then it follows that the whole software would have to be re-examined in order for the developer to rebuild their understanding. During this process, small changes may go unnoticed, with the developer believing instead that no changes had been made.

Therefore, the clear identification of changes is an important weapon in a developer's armoury. By identifying the exact change that was made to the source code, it is possible for the developer to update their understanding of the software to account for the change. However, the increased complexity associated with software evolution means that the effect of a change may be wide ranging, and unrestricted to a specific function or file. Therefore, in order to provide the maintainer with a clear picture, it is necessary to provide not only the actual changes made, but also the impact that those changes had across the project. This is difficult to achieve with text-based tools such as ViewCVS [ViewCVS03] (Figure 1-1) that are often provided.

1.1.3. Software visualisation

Visualisation aims to enhance the comprehension process of a user, and allow the user to view the problem from different perspectives, using different cognitive skills. The use of diagrams to represent information has been well analysed, and in many cases has been shown to be more effective than displaying the raw data. Information visualisation is popular amongst scientific and financial communities, where the level of data is such that looking at raw figures is often meaningless. Representing this data graphically allows patterns and trends to be identified that would otherwise remain hidden.

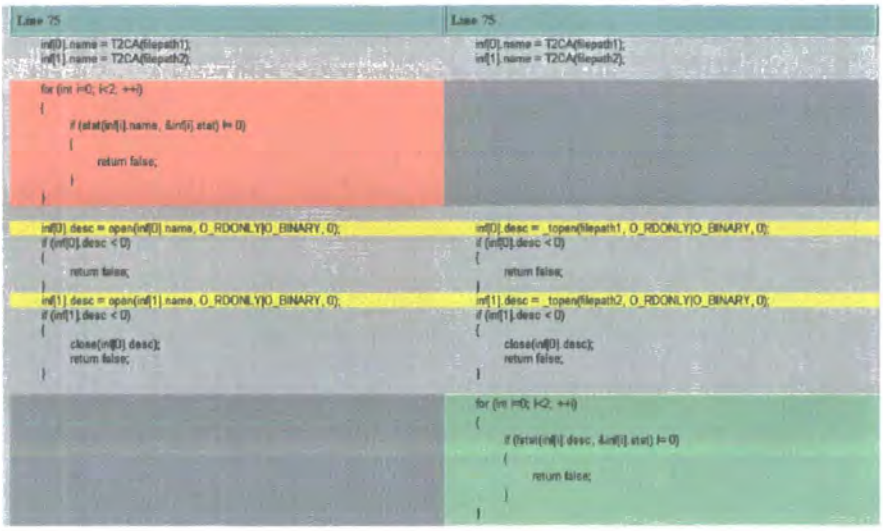


Figure 1-1. Differences between two versions, shown using ViewCVS.

Software visualisation is the application of information visualisation to software. This is often done graphically, with nodes representing the entities within the software, such as modules, files or functions, and arcs representing relationships between them. Such a representation has problems of scale when large numbers of entities are involved. Therefore, recent research has concentrated on other forms of representation and introduced new interaction techniques allowing the user to abstract or filter unwanted information.

Software visualisation research has concentrated predominantly on 2D representations, with more recent research examining the possibilities of the use of 3D and virtual reality environments. In general, these visualisations aim to represent the state of the software at the current point in time, at differing levels of granularity. However, there has been little research into visualising the state of the software over a period of time, and thereby showing how the software has evolved.

1.2. Objectives

Some of the problems associated with open source development and software evolution in general have been mentioned above. A solution to these problems may lie within visualisation. However, existing software visualisation systems are designed predominantly to support only static data, and so changes may only be identified by comparing two adjacent displays representing the visualisations of the 'before' and 'after' states.

Research has indicated that there is a vast amount of knowledge contained within change management systems, which may allow patterns and trends to be identified in the evolution process. However, as Dijkstra [Dijkstra68] notes, extracting this information is difficult. He writes that:

> *"our intellectual powers are rather geared to master static relations and that our powers to visualise processes evolving in time are relatively poorly developed. We should do our utmost to... make the correspondence between the program and the process as trivial as possible".*

Therefore, using visualisation to highlight changes has two benefits. Firstly, it allows a developer to identify a change within the context of the whole software, and so have a better understanding of the effect of that change within the system. Secondly, it reduces the intellectual powers required in order to determine the long-term effect of a sequence of changes, and therefore allows a developer to examine the success, or otherwise, of these modifications. This also allows trends within the software evolution process, such as specific files undergoing a far higher level of maintenance than others, to be observed. The focus of this research is therefore to develop visualisations that achieve this.

However, visualising historical data raises new problems within software visualisation. As the software continues to evolve, the amount and complexity of data to be visualised will also increase. The new data, representing the most recent modifications, will then need to be incorporated into the existing visualisation with minimal impact. This research will also investigate the ways by which this may be achieved.

Animation is a powerful means of conveying temporal information. For example, an animated sequence showing continental drift or population shifts in a geographical system, or a chemical reaction in a chemical system can show much more information than many equivalent static diagrams. This has also been shown true within software, by animating the execution and effects of simple algorithms. However, software visualisations are restricted predominantly to 2D or 3D representations. Therefore, due to the temporal nature of software evolution, this research will examine the effects of using animated representations within software visualisation.

1.3. Criteria for success

The overall criteria for the success of this research may be considered to be the development and evaluation of visualisations that allow the differences between releases of continuously evolving software to be identified.

This may be broken down into a number of areas, which will be addressed by this thesis. The criteria for success are therefore:

a) Identification of the benefits of visualising evolving software.

Very few examples of visualisations of software evolution exist. Therefore, before developing additional visualisations, it is necessary to identify the purpose and relevance of visualising software evolution in order to determine whether these would be beneficial.

b) Identification of the key aspects for visualising evolving software.

Evolving software presents a number of new issues within software visualisation, such as a regularly changing data set. These issues must be identified in order to produce successful visualisations.

c) Assess the suitability of animation within software visualisation.

Animation is a natural means of showing progress through time. However, there are few examples of the use of animation within software visualisation. Therefore, the possibility, feasibility and implications of using animation should be investigated in order to demonstrate whether using animation is a viable approach.

d) Development of new visualisations highlighting software evolution.

Visualisations will be developed and evaluated to solve the problems identified within this introduction. The visualisations must consider the results from b) and c).

e) Address issues of scalability to be suitable for real world projects.

In order to be useful and relevant to software evolution, the visualisations must be able to support actual projects, rather than only being useful for artificially small projects.

f) Assess the feasibility of automatic generation of the visualisations.

Within software visualisation, a hand crafted visualisation is viable provided that the time to create the visualisation and comprehend it is less than the time that it would have taken to comprehend the software without the visualisation. However, a hand crafted approach is not feasible when examining software evolution. The continuous evolution implied by open source development in particular would more than likely result in the manual visualisation being out of date before it was completed. Therefore, it is necessary to demonstrate that the visualisations may be created automatically.

These criteria will be evaluated in chapter 8.

1.4. Thesis Overview

This thesis is composed of eight chapters, of which this is the first. Chapter 2 provides an overview of existing research within four separate areas, and acts as a foundation for the remainder of this research. The research into software evolution, configuration management, software visualisation and animation is summarised, before the chapter concludes by identifying the overlap between these areas.

Chapter 3 introduces the idea of evolutionary software visualisations, as visualisations targeted specifically towards historical data. The difficulties of evolutionary visualisations are described, and a number of solutions are proposed. These solutions are based on layout, representation, and the use of animation.

Chapter 4 introduces the first of two evolutionary visualisations that focus on identifying change within software. The visualisation concentrates on configuration management data, and demonstrates the use of some of the techniques from Chapter 3.

Chapter 5 introduces the second visualisation. This visualisation identifies change at the source code level, and demonstrates other techniques developed within Chapter 3.

Chapter 6 provides a summary of the implementations of these visualisations, in order to demonstrate feasibility and automation. The process of generating the visualisations, and the tools and techniques used are also covered.

Chapter 7 evaluates the visualisations from Chapter 4 and 5. A short summary of evaluation techniques is provided, and then the visualisations are evaluated using four methods. The chapter contains a critical, informal evaluation of both visualisations, before these visualisations are evaluated formally using existing frameworks. The features of the visualisations compared to existing tools are also analysed. Finally, a number of usage scenarios are described illustrating relevant change management tasks.

Chapter 8 provides a summary of the results of this research, and some of the conclusions that may be drawn. This chapter also re-examines the criteria for success listed in section 1.3. Finally, areas of future research are outlined.

ANIMATING THE EVOLUTION OF SOFTWARE

2. Current Research

2.1. Introduction

In order to develop visualisations that are suitable for examining the evolution of software, it is necessary to have a clear understanding of several related research areas. This chapter will summarise existing work in four fields: software evolution, configuration management, software visualisation and animation. The first of these, software evolution, covers how software is modified and adapted over time as the requirements of the users change. This section details why the changes are necessary, and the types of change that can occur. It also examines the extent to which the evolution may be measured and predicted. Finally, some results from research into the evolution of open source projects are included in order to compare open source and traditional closed source development techniques.

Configuration management, and particularly version control, is related to software evolution. Configuration management solutions allow the historical information associated with evolved software to be stored and retrieved when necessary. As well as including the source code, this data may also include documentation, test data, and fault reports. Additionally, due to the large amount of data involved, it is necessary for these systems to minimise the storage space required as the software evolves. This is achieved through the use of deltas, where only the differences between subsequent versions are stored. Examining these deltas therefore allows the changes between versions to be identified.

The extent of the data contained within a configuration management system means that it is difficult to comprehend. The problem of understanding this data may be resolved using software visualisation. The third section examines a number of existing guidelines and techniques that may be relevant for displaying this data. In particular, the current solutions for visualising different aspects of software evolution are analysed. Finally, animation is a natural means of showing time-based information. Therefore, the final section summarises the existing use of animation within software and information visualisation, in order to determine whether any benefit may be derived by using this approach.

2.2. Software Evolution

2.2.1. History of software evolution

The concept of continual program growth was identified by Lehman in the late 1960s [Lehman85]. In order to model this process, the idea of program growth dynamics, and later, program evolution dynamics, was introduced.

Although the term 'software maintenance' is common, it is sometimes considered inappropriate [Lehman85,Cook90]. Maintenance implies that there is a deterioration of some object over time, due to internal or external influence, and it should therefore be restored to its initial state. However, with software, this is not the case. Provided that the environment that the software is placed in remains *identical*, the software will remain as useful as ever. Instead, the term 'evolution' suggests that the

software is adapting continually to an ever-changing environment, which is a more accurate reflection of the process.

Moreover, this process is inevitable. As Lehman writes:

> *"It must now be accepted that evolution is, ultimately, not due to shortcomings in current programming processes. It is intrinsic to the very nature of computer usage; computing applications and the systems that implement them"* [Lehman85].

Initially, software evolution was seen originally as a single end stage of a traditional software lifecycle of requirements gathering, design, implementation and testing. However, this idea has since been expanded by Rajlich and Bennett [Rajlich00] to include servicing and a phasing out stage. Servicing arises when further evolution is no longer cost-effective, and results in very few changes normally implemented as wrappers. The phase out stage occurs when servicing is withdrawn but the owners still seek revenue from the software. However, as these final phases are as a direct result of management influence, it is the main evolution phase that is of most interest to developers.

2.2.2. Evolution studies

A number of studies have been implemented in order to examine why and how software evolves. This section summarises the results and conclusions of some of these.

2.2.2.1. Why software evolves

Over many years, Lehman formed and refined eight laws of software evolution [Lehman00]. These laws were derived from careful study of several industrial sized systems, and have since been confirmed by the FEAST/1 and FEAST/2 projects. [Lehman01] They provide an invaluable insight into software systems, and provide a basis for understanding why and how software evolves.

The laws are based on 'E-type' systems, or systems that are embedded in an operational environment. A defining factor of these systems is that they will never have a complete, satisfactory specification, as the variety of features that can be added to the systems is unlimited. [Lehman85] The following laws are interdependent, and so should not considered to be linearly ordered. In addition, Lehman also suggests a number of guidelines in order to manage the impact of these laws. [Lehman00]

1. **Continuing Change – E-type systems must be regularly adapted else they become progressively less satisfactory in use**. This is due to feedback from the operational domain, which changes once the software is used. A new software release is then necessary when, for example, assumptions made within the current release become invalidated. Due to this, Lehman suggests the need for comprehensive, regularly maintained documentation that records new content and interdependencies. Also, design decisions need to be identified within the documentation, together with the assumptions that those decisions are based on.

2. **Growing Complexity – As an E-Type system is evolved its complexity increases unless work is done to maintain or reduce it.** As changes are applied to the system, the dependencies between components in the system increase, leading to greater complexity. It is suggested that the growth of complexity is measured in order to determine when reengineering strategies are required.

3. **Self Regulation – Global E-type system evolution processes are self regulating.** Large commercial products will be implemented by a large team, and often in a larger organisation. The organisation has interests far beyond a single product, and so every project is subject to the same regulations and checks. The growth then becomes driven by these regulations and so establishes its own dynamics. Often, growth across different projects is surprisingly similar – within the Feast project, it was common for the ratio of the number of modules within a project to the number of releases to map well onto an inverse squared relationship. Unexpected increases in size against this model generally cause problems such as low quality or late delivery of the system.

4. **Conservation of Operational Stability.** Increasing the resources available to a project does not necessarily improve the activity rate, due to the increased overheads of communication. The activity rate is also dependent on system attributes, such as complexity, which is driven by the third law.

5. **Conservation of Familiarity – In general, the incremental growth and long term growth rate of E-type systems tend to decline.** This refers to the fact that the more changes that are associated with a particular release, the more difficult it is for those involved to understand the impact of these changes. It therefore takes longer before the system can be maintained reliably again. Also, as comprehension comes at an ever-increasing cost, and assuming that the budget is constrained, fewer resources are then available for new functionality. In order to manage this, it is suggested that change data should be plotted to determine evolutionary trends. This includes changes in objects, lines of code, features, requirements, and other such artefacts. Change logs should be updated with a fixed template in order to facilitate later data extraction. This data should be examined regularly to find the clearest indication of evolutionary trends so remedial action may be taken if necessary.

6. **Continuing Growth – The functional capacity of E-type systems must be continually increased to maintain user satisfaction over the system lifetime.** When software is developed, the user is not usually aware or able to have the exact system that they want delivered. The missing functionality will soon be identified when the software is used, and human intervention is required. Competition may also lead to new features being required.

7. **Declining Quality – The quality of E-type systems will appear to be declining unless they are rigorously adapted, as required, to take into account changes in the operational environment.** There is always a gap between a finite software system and the real world problem. Constraints are added to the system in order to restrict the problem to one that may be implemented. As the environment in which the program is based however, some or all of these initial constraints become invalid, leading to inaccurate behaviour. Furthermore, the increase in

complexity means maintenance becomes more difficult, with an increased likelihood of introducing faults. In order to reduce this, Lehman suggests the use of programming principles such as implementation hiding to reduce the impact of changes that are made, and also the provision of resources for restructuring and removal of 'dead' code to ensure that time is not wasted maintaining it.

8. **Feedback System – E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.** Feedback includes that caused by the ripple effect of changes and the problems of excessive growth in the fifth law. Feedback is unavoidable in a software system, leading to long range trends and invariants. These must be observed in order to modify the system in such a way as to maintain stability.

These eight laws explain why software evolution is such a critical part of the software engineering process. However, it is a difficult and time-consuming task. Schneidewind [Schneidewind87] states that the process of maintaining software is hard because of the poor documentation and predicted impact of changes. Brooks [Brooks87] suggests that software has inherent properties of complexity, conformity, changeability and invisibility, all conspiring to make understanding and modifications difficult.

Scheniedewind also suggests how maintenance should be improved through better designs, implementations and methodology. However, in a real world scenario, it is unlikely that such a process is frequently followed [Humphrey88], either due to capital or time constraints, or poor education. Even if software developed now followed these ideas for better maintenance, there are many systems still in existence that will continue to be used until the resources are no longer available to maintain them.

Regardless of the actual change to be made, it is necessary that maintainers have some understanding of the program, whether at a source code, design or requirements level. Program comprehension therefore is probably the key aspect of software maintenance. Many comprehension strategies exist, which have been combined together by Mayrhauser [Mayrhauser95] to create a comprehensive meta-model. However, regardless of the strategy employed, comprehension is a difficult, error prone, and time consuming task.

2.2.2.2. Changes that occur during evolution

Traditionally, changes that occur within the maintenance stage may be categorised into three or four types. These may be defined as: [IEEE94, in Mattsson00]

Corrective: Maintenance performed to correct faults in hardware or software.

Adaptive: Software maintenance performed to make a computer program usable in a changed environment.

Perfective: Software maintenance performed to improve the performance, maintainability or other attributes of a computer program.

Preventative: Maintenance performed for the purpose of preventing problems before they occur.

Various studies have taken place in order to determine how much time and effort is required for each maintenance task.

Mockus [Mockus00a] developed a system to automatically classify the type of maintenance that was being carried out. This was based on examining the comments contained within modification requests based on a 2MLOC system. Three types of maintenance were examined: corrective, adaptive and perfective. Corrective maintenance was identified by words such as 'fix', 'problem', 'correct' and 'incorrect'. Adaptive maintenance was identified by the words 'add', 'new', 'modify' and 'update'. Finally, perfective maintenance was detected by the words 'cleanup', 'unneeded', 'remove' and 'rework'. Various rules were applied in order that each request was uniquely classified. The results indicated that 45% of all requests were attributed to adaptive maintenance, and a third to corrective. 12% were unidentified, but the majority of these were also thought to relate to fault fixes from further analysis. Developers were also asked to classify requests as 'easy' 'medium' or 'hard'. A corrective maintenance request was found to be significantly more difficult to implement than a perfective request, with adaptive requests considered to be the easiest. By basing the system on a version control repository, it was also possible to determine the length of time required to implement the different types of change. Fault fixes could be done in the shortest time, followed by code improvements and then code development. Therefore, code development was seen as easy but time consuming, whereas fault fixing was seen as quick, but more difficult.

Jørgensen [Jørgensen95] in a study of a large software system examined four maintenance types. 28% of tasks were seen as adaptive, taking 40% of the maintenance effort. 38% of tasks were classified as corrective, taking 9% of effort. 29% of tasks were identified as perfective, requiring 45% available effort. Finally, 5% of the required tasks were recognised as preventative, requiring 6% effort.

The results between these studies are reasonably similar. Slightly more development was required in the Jørgensen study – 57% opposed to 45%. Less corrective maintenance was also done – 38% as opposed to 46%. The effort required was also similar, with corrective maintenance taking much less time than code development.

In order to resolve some of the current confusion within the field as to the precise definitions of the maintenance types, research has suggested a significant expansion of available maintenance categories from the traditional three or four to twelve [Chapin00]. Within this new framework, maintenance activities are clustered into one of four categories: support, documentation, software properties, and business rules. This provides a much stricter definition of individual maintenance types. For example, changes to documentation are given two new terms. Seven terms are allocated to maintenance relating to a source code change, including the new terms 'groomative', 'reductive', 'performance' and 'enhancive'. The result is a clearer definition of the maintenance tasks, and removes the overlap and uncertainty between corrective and perfective, or perfective and preventative for example.

2.2.2.3. Measuring evolvability

Cook et al [Cook00] define the concept of evolvability as "the capability of software products to be evolved to continue to serve its customer in a cost effective way". This is an important consideration when examining evolving software as it determines when there is a need for preventative maintenance, or instead, when the software should be phased out. Evolvability may be assessed to some extent quantitatively by measuring 'code decay', defined as a measure of the extent to which "code is more difficult to change than it used to be" [Eick01]. Highly decayed code will be more difficult and costly to maintain. Various recent studies have addressed code decay and evolvability from different perspectives, either by examining the software architecture, or predicting future fault rates and effort requirements from the current state of the software.

Two software evolvability metrics were proposed by Burd and Munro [Burd99]. They hypothesised that the call structure within the software is particularly important when maintainers need to understand the existing code. This hypothesis is based on observations that maintainers were more unwilling to change the call structure than data structures when maintaining a program. The argument follows that if the call structure has been significantly changed then there must have been a significant amount of maintenance performed on the code. By analysing this call structure, an assessment of whether the code is degrading or improving can be obtained. The measurements were then verified against two versions of the compiler GCC, along with examination of change logs and interviews with maintainers of the program. All the methods used indicated the same result – that of a large amount of preventative maintenance occurring between the two versions. However, additional evidence of the accuracy of the metrics for other projects is not available.

Ohlsson et al [Ohlsson99] used various historical data in order to classify components on a scale of 'fault-proneness', representing the extent of code decay. Based on a case study of 800KLOC of C, 28 different metrics were collected. Most of these were based on the number of files that had to be modified in order to implement the change, and the number of times the individual files were modified. Header files and implementation files were considered separately, as these were thought to have very different properties. Any increases in file size and coupling were also recorded.

Healthy components could usually be identified as ones where any required changes occurred primarily within the implementation files, and additionally where these changes could be implemented relatively easily. Interestingly, low coupling was found to be insignificant. Problematic components involved a larger number of defect reports, a larger number of affected files when implementing change, and often the need for external components to also be changed.

Graves et al [Graves00] also assessed code decay from a fault perspective. A number of models were proposed. The initial model proposed that the number of faults in a future module was based on the number of faults that had occurred within that file in the past. An improved model examined the average age of the code, and the number of required deltas to make a change. The most successful model however was one that was time based. Fault potential was computed by adding the contribution of individual changes, weighted towards large and more recent changes.

Interestingly, several factors were also found to be poor predictors of faults. The size of the program in terms of the number of lines of code was not found to be relevant. They also found that complexity metrics had a high correlation to lines of code, and so these too were not thought to be useful. Also, the number of developers that had made previous changes to the module appeared to have no effect. Finally, as with Ohlsson's study, the extent to which modules were changed concurrently with other modules was not significant.

Graves and Mockus [Graves98] carried out a similar study, based on a large telecom switching system. Evolvability was determined by monitoring the time taken to implement a change. Experienced developers, defined as working on the project for several years, were used as participants in the study. Data was taken from a four-year period of development, and analysed. Four variables were found to be critical to determining the change effort required. The first was the size of the change, which was measured as the number of deltas required for the change to be complete. The second was the type of change, classified using comments within the maintenance requests [Mockus00a]. The third was the actual developer making the change, with some developers requiring less effort to implement changes than others. Finally, the time of change was found to be significant. It was determined that within their study, a change begun a year later than an otherwise similar change would require 20% more effort.

Finally, Eick et al [Eick01] also addressed this issue using the same telecom switching software as above, and also attempted to determine some of the causes of code decay. Many of the reasons given are similar to those proposed by Lehman. Other examples given included pressures such as cost or time that affect developers resulting in shortcuts taken during implementation, and poor initial architecture making some changes significantly more difficult. The importance of tool support was also raised, with poor CASE support or version control systems causing difficulties.

Symptoms of these causes included bloated code, a history of faults and frequent changes, and widely dispersed changes. The presence of a large number of interfaces into a module was often cited by developers as being problematic, as the increased number of interfaces meant a higher likelihood of side effects.

A number of risk factors were also identified implying the increased likelihood of code decay. These included the size of the code, with large modules likely to contain more symptoms. The age of code was also important. Old code may be more stable with all the faults removed over time. However, at the same time, new code will be developed for the current environment and so will not have the problem of obsolete assumptions. Therefore, the age variability within a module was considered to be most important. Porting and reusing code was declared as a risk, as assumptions made within one system may be invalid for another. A large number of requirements within a project also means that the software is subject to many constraints that must all be satisfied. These can prove difficult to understand and implement. Finally, inexperienced developers increase the risk of code decay due to a lack of knowledge of the system and less developed skills.

Based on these factors, a model for estimating the required effort when making changes was developed. The resultant effort required was based on values for the number of files affected by a change, the number of lines added and deleted, and the number of deltas required to make the change.

It is clear from these studies that there are a number of factors that affect the evolvability of software. For example, the number of deltas required to implement a change appears to be a good overall indicator of the state of the software. Once identified however, this research highlights that improving the evolvability is both difficult and expensive. For example, Burd and Munro concluded that improving the calling structure – and so possibly the relationship between files - was a sign of preventative maintenance and thus improved evolvability. However, such a change will result in many files being affected, and so require a large number of deltas. This will result in an increase in both the effort required and the risk of faults when implementing the required modifications. Once the modification has been made successfully though, future changes should be both simpler and less fault prone.

2.2.3. Open source studies

2.2.3.1. Open source software development

The term 'open source' was coined in 1998, when Netscape decided to release the source code of their web browser to the public, in order to compete better with Internet Explorer [OSI03]. As the name suggests, open source software is software that is provided with all source code that is required in order to build and use the product. This code may then be distributed and modified without charge. However, depending on the licence agreements of the original software, the changes made may also have to be released freely. Additionally, in some cases, software developed using open source software may also have to be released under an open source licence. Although the term 'open source' is new, the principle dates back much further [So02].

Open source development is often considered to be a completely different method of development from a typical software lifecycle [Raymond99]. It is usually based around a small core of regular developers who do the majority of the development [Mockus00c]. However, their role is made easier by a large number of contributions from other, more informal developers. As the code is accessible to everyone, these contributors are capable of examining the code in order to fix faults and suggest further modifications. These can be provided in the form of suggestions, or as is often more likely, source code patches with the required functionality. In order to ensure that everyone is working from recent code, it is important that releases are very frequent – far more often than with traditional development methods [Raymond99].

As the software is available freely, projects can only afford to employ a small number of developers, if any. Therefore, many of the contributions provided will come from volunteers. Given that the process

relies heavily on these volunteers, it is important to examine their motivation for being involved with a project.

There are two types of contributors [Gaceck02]. Corporations will get involved either to gain market share and undermine their competitors, or because they have source code access to suitable products for free without having to develop their own in house. Kenwood [Kenwood02] suggests some of the benefits and drawbacks of this approach, considering aspects such as cost, availability and security. Individuals will usually contribute code for a variety of reasons [So02]. It may be because they see that the open source software provides a good solution to their needs. Alternatively, it may be due to social factors – reciprocal behaviour, reputation or attraction to the community. In some cases, particularly regarding the Free Software Foundation [FSF03], idealism is also a significant motivator. Contributions can provide other benefits, such as the opportunity to develop new skills, or the opportunity for collaborative programming.

However, there are a number of reasons why volunteers may find it difficult to participate in a project, mainly stemming from the fact that the developers will rarely meet [So02, Reis02, Yamauchi00]. Firstly, they may not have the time to continue working. Secondly, there is also a significant linguistic barrier, with most projects being developed in English. As email and newsgroups are often the primary communication medium, the ability to write clear English is vital. Finally, the complexity of the source code may prove difficult, particularly if the software is designed badly or has poor documentation. The documentation aspect is a serious problem, particularly for new contributors, as the documentation is often seriously outdated, and general overviews difficult to obtain. In addition, more tool support is required. As Reis writes,

> *"The lack of tools to aid communication and visibility into the process can seriously hamper the project's progress, so their availability, quality and usability are of dire importance".*

Open source development is not appropriate for every situation. In particular, one of the crucial differences between this and the traditional model is that the developers are also users of the software [Raymond99]. This means that technical, domain specific areas such as medicine or air-traffic control would have a very low potential success rate if an open source strategy were applied. This is because the majority of the users of such a system are unlikely to have the skills required to develop the system further. A benefit of this however, is that developers are only working on things for which they have a passion, resulting in more care and creativity when coding [Raymond99]. However, Kenwood [Kenwood02] notes that the developers that do participate tend to focus on fellow technical users, leading to relatively unfriendly user interfaces with minimal help support. Also, it is important that the project is large and interesting enough to attract a large number of contributions [Kenwood02], as this leads to more development [Koch00]. Often, this may be achieved by releasing a mature closed-source project as open source, rather than developing the entire project from the beginning.

Projects based on open source software development will often have other common features. Godfrey and Tu [Godfrey00], and Mockus [Mockus00c] both observe that there is usually no set time frame for releases, and therefore the developers are not so constrained by deadlines. This may lead to releases of

more stable and mature software. Additionally, detailed test plans are also less likely than with closed source development, particularly in smaller projects. These are instead replaced with the 'many eyeballs implies shallow bugs' [Raymond99] approach to fault finding and fixing, known as Linus' Law. This refers to the fact that, as the code is open, more people have the ability to look for bugs simultaneously.

2.2.3.2. Case Studies

As open source development is different to that of traditional closed-source development, it is worth examining the process in more detail. There have been a number of recent studies that have done this, examining many of the open source success stories.

Godfrey and Tu [Godfrey00] examined 96 versions of the Linux kernel in order to measure growth over a period of six years. Due to the 'less structured and less carefully planned' evolution of open source software, they expected that Lehman's laws would apply, with growth approximating an inverse square law. Growth was measured based on uncommented lines of code. Interestingly, they found that growth was accelerating over time. Further analysis of the subsystems within the kernel was carried out in order to try and determine how this growth rate had been achieved. This revealed that over half of the code was based on implementing device drivers, and so was relatively uncoupled. Therefore, this suggested that parallel development could take place without introducing further complexity. They concluded that this allowed the growth rate to remain constant, rather than diminishing with time. The super-linear growth rate was not explained however.

Koch and Schneider [Koch00] concentrated on the GNOME project – an open source project providing a desktop environment and application framework. Data was based on mailing lists, bug tracking archives, and the source code retrieved from CVS. Metrics were generated from the LOC added to a file, the number of check ins made to the project and the time spent on the project by each contributor – determined by the difference between the first and last contributions. Over 300 developers added a total of 6.3MLOC over the course of the project, and deleted 4.5MLOC. The large number of deletions is probably a result of CVS recording a changed line as an addition and deletion. Results showed that the majority of code was produced by a smaller number of developers. Further analysis showed that programmers adding more LOC used more check-ins, rather than submitting larger patches. However, the size of the patch was influenced by the size of the file, with large patches being checked in to large files. The software was growing linearly, although after two years this could be consistent with an inverse square model. However there is currently no hint of slow down.

Mockus et al [Mockus02] examined two further open source projects – the web server Apache and the web browser Mozilla. In addition to observing the projects individually, comparisons were made between the fully open source development methods of Apache, and the commercial hybrid of Mozilla. Comparisons were also made to commercial software. A number of hypotheses were proposed after examining Apache, and then tested against Mozilla. The number of contributors was examined in both cases. With Apache, 182 people contributed to 695 fixes, and 249 people contributed to 6902 code

submissions. 3060 people submitted 3975 problem reports, although only 500 reports from 450 users subsequently resulted in a change to code or documentation. The top 15% of developers contributed well over 85% of any changed code, although only 66% of the fixes. In addition, the core developers submitted only 5% of the problem reports, a clear indication of the reliance on external contributions for system testing. Mozilla is approximately ten times larger than Apache. This was reflected in the number of contributions. They observed that about 6800 people generated 58000 problem reports, although only 11500 of these, from 1400 people, resulted in a change. Although far more reports were generated per person in Mozilla, the proportion of reports resulting in changes to the code was similar in both studies. In addition, nearly 500 developers contributed code, and 400 contributed fixes. However, as found within the Apache study, a small core of developers did most of the new development.

The hypotheses presented by Mockus et al suggest that a core development team will create approximately 80% of the functionality, with around 10-15 people unless more formal development processes are implemented. A group significantly larger than this will repair defects and an even larger group will report problems. Defect density will also be lower than commercial code receiving equivalent testing, with a rapid response to any customer problems. These hypotheses also seem consistent with the study by Koch.

These studies are over a relatively short period of time, and so some of the long-term implications are not yet apparent. However, indications would suggest that at least some of Lehman's laws regarding growth are less apparent than in traditional software development methods. A possible reason for this is the continual increase in contributors, for the reasons specified earlier. As the projects become better known and more complete, increased publicity means that this should continue. With traditional development, adding large numbers of developers to a project will not normally aid the process due to the significant communication overheads. However, within open source development, many of the submissions made by the contributors are independent of any other communications. Also, the core development team still remains about the same size. The benefits of contributors come from the time consuming tasks of fault finding and fixing. By leaving this to the community, core developers can spend the majority of time introducing new functionality.

2.2.4. Summary

This section has summarised some of the existing research examining software evolution. Many of the underlying reasons why software evolution is an inherent aspect of software development have been identified, illustrating why the difficulties of continued evolution can not be ignored. Secondly, a number of methods have been listed to indicate the extent to which it is possible to predict the ease of making modifications to the software to allow it to continue to evolve. Finally, some of the recent research examining open source development, as this has a large emphasis on evolution, has been summarised. The benefits of open source software, such as a lower number of faults, and a faster response when faults are identified, are significant. However, in order to achieve this, it is necessary to

increase the number of developers that are prepared to participate in projects by removing some of the barriers, such as poor documentation, that may otherwise prevent them from joining.

2.3. Software Configuration Management

Software configuration management (SCM) can be defined as:

> *"... the controlled way of leading and managing the development of and changes to combined systems and products during their entire life cycles".* [Asklund01]

This definition demonstrates the importance of using a software configuration management process within software development, and software evolution in particular. This section will examine the features required by configuration management systems, and analyse a number of existing systems. Finally, the means by which changes may be determined is addressed.

2.3.1. Features

Estublier [Estublier00] proposes a number of major aspects of a SCM system. Firstly, a component repository is required to store all the objects within the software development environment, including source code, documentation and test data. These objects will be versioned by creating a new revision each time the file is changed. These revisions are usually identified by a series of numbers to allow later retrieval. Where necessary, new branches may be created, allowing the same file to be changed in different ways. The versioning system also needs to consider history, deltas and user management to allow better management of the process. Provision of historical information may be easily achieved by recording changes within a log. Deltas are used to reduce the storage space required by the repository, by only storing the changes between two revisions. User management ensures that two users do not modify the same file simultaneously. This is usually implemented as a locking mechanism, although other solutions also exist. The repository also needs to consider various configurations of the objects – "a set of files, which together constitute a valid software project" [Estublier00]. Configurations may exist permanently within the repository, or may be generated when necessary by applying various change sets to a base configuration.

Engineer support is provided through build and workspace support. Build management handles the building of the software to produce the final executables. Where possible, this should reuse as many valid objects as possible, and only rebuild modules that have undergone changes – either directly, or as a result of a dependency. Make [Make03] is a very successful tool for achieving this.

Workspace support provides a developer with easy access to the various files stored within the SCM system. Essentially, the effect is that of a time machine, where the environment may be set up to display the complete status of the project at any point in time [Leblang94]. Ideally, this process should be completely transparent. This is achievable through the use of a virtual file system such as VCFS [Chee97].

Finally, process support is required in a successful system. This encompasses many other areas, such as security and reliability. Security options may provide restricted access to various objects within the system, ensuring, for example, that developers are limited to working on individual modules within a larger project. Methods for improving reliability may include monitoring the check-in procedure to ensure that the code satisfies certain complexity metrics, or the provision of support for defect tracking.

2.3.2. Tool support

It is clear that successful configuration management will require good support tools. One of the first tools to be developed was SCCS, which was later succeeded by RCS [Tichy85]. These tools only satisfy the version control aspects of a configuration management system and are implemented using a simple file-based repository. In order to modify a source file, the tools require the file to be extracted from the repository and placed in a private workspace. The file can then be modified as necessary. Finally, the tool will store the modified file in the repository. If the source file was not locked, allowing other developers to also modify the file in parallel, a merge operation may be required when the file is checked in.

RCS has a number of problems, such as poor merge and branch capabilities, no binary file support, and no change grouping [Leblang94]. Transparency is also an issue. This is because the files are stored in a central repository and so it is not possible for a software tool to access them directly. Therefore, in order to compile the software, it is necessary to extract all of the necessary sources from the vault, including those that do not need to be altered. Various enhancements have been made to solve some of these problems. For example, PRCS [MacDonald98] allows files to be grouped into projects and RCE [Tichy03] provides support for binary files.

Far more complete tools for SCM exist than RCS [Estublier00]. For example, ClearCase [Leblang94] has integrated support for many of the required SCM features suggested by Estublier and is popular for closed source development projects. However, the situation is different for open source development.

CVS [Berliner90] is a multi-user SCM tool based on RCS. The interesting aspect of CVS is that although it is much older and less complete then tools such as ClearCase, and has several problems which are similar to those encountered with RCS, it is almost synonymous with open source software development [Asklund01, Reis02, Hoek00]. Hoek suggests three reasons why this is the case.

Firstly, CVS closely mirrors the typical open source development process. Open source projects are based round a central point of contact, containing access to the repository, forums, fault reporting functions, and so on. As developers are usually voluntary, they are unlikely to have, or require, continual access to this repository. Therefore, the process of downloading, making changes locally, and then uploading the new files required by CVS is more than adequate for open source development. Secondly, CVS supports a distributed environment, which is vital for open source projects. Finally, CVS itself is provided under an open source licence, and is available freely. As open source development usually has little funding, the use of an expensive commercial SCM system is unlikely. Hoek goes on to suggest various enhancements that could be made to CVS to make it even more

suitable for open source development, including improved versioning capabilities, and better change and release management.

Primarily as a result of Mozilla, new tools have been introduced that provide CVS with some of these enhancements. BugZilla [BugZilla03] is a crucial tool within larger open source projects, allowing complete control of change requests [Reis02]. Changes can be classified with various priorities and severity, as well as whether they are affected by other required changes. They can also be assigned to a developer who can indicate when the change should been completed. Most importantly for the open source community, a series of comments can be made about the change. This also allows the ability to provide patches for fixing the bug or adding the feature, which then go through a visible review process. It is now also used in many other major projects such as Redhat Linux and the Gnome desktop environment.

Bonsai [Bonsai03] provides another useful extension to CVS by providing a web based interface into the CVS repository. As well as showing comments within the log files, together with a reference to the change within BugZilla, it also highlights differences between versions. In particular, the ability to see visually which developers are responsible for which lines in a file is very useful for project management purposes.

These extensions, and others, to CVS have enabled it to resist competition from other, more complete tools. The dominance it holds within the open source community also means that it is unlikely to be replaced in the near future.

2.3.3. Identifying differences

As stated earlier, an important part of any version control tool is the ability to identify differences between two versions of the source code. This allows a delta to be produced containing just the differences between the two files, and so reduces the required storage space for the repository. Deltas may also be used for other operations. For example, in open source development, a patch may be submitted as a delta, rather than a complete file. This again reduces the storage requirements, but also means that if the file has changed since the patch was developed, those new changes will not be lost unless they conflict with the delta.

Identifying differences is also important from a maintenance perspective. Yang [Yang91] claims that:

> *"Accurately identifying the differences between program versions helps the maintainers understand the programs and eases the maintenance task".*

The theory is that if the maintainer has a good understanding of the previous version of the software, then highlighting the changes clearly will help them to update their knowledge of the software. If these changes are not clear, then far more effort will be spent in this process. The importance given to this may also be assumed from the large number of diff-based front ends that exist which seek to display the changes visually, such as CsDiff [CsDiff03] and SeeDiff [Ball96].

2.3.3.1. Text based

Tools to generate deltas are therefore valuable to both configuration management and software maintenance. The Unix tool 'diff', based on an algorithm by Hunt [Hunt77], and used in RCS [Tichy85] and VCFS [Chee97] to name but two, is a popular choice. (Other similar tools exist, such as FComp [Miller85]). Diff generates deltas recording where entire lines in a text file are different, rather than individual characters. It was thought that a character based delta 'would take much longer to compute, and not be significantly shorter' [Tichy85]. However, the problems with using diff have been well documented, by Hunt [Hunt98] amongst others. Two main difficulties in particular should be highlighted.

The first is that, due to the line-based delta approach, diff deals very badly with binary files. As it is often useful to store such files, such as design diagrams or word-processed documents as part of a revision, this is a serious failing. The standard solution is to encode the file into ASCII and allow diff to process the encoded version. However, the encoding process must also insert line breaks into the file. In this case, a single byte inserted or deleted could mean that every line is recorded as being different. The second issue is that interchanging of lines is not recognised. For example, if a section of text is moved from one part of a document to another, diff will record this as a deleted section, and later a new section.

To combat these problems, new algorithms have been developed. Bdiff was developed and incorporated as part of RCE - an enhanced, commercial version of RCS [Tichy03]. The significant difference is that Bdiff generates byte-based, rather than line-based deltas, and thus is suitable for binary files. Another algorithm - VCDiff [Korn02] is based on a compression algorithm, and solves the problem of interchanging lines as well as binary files. However, these improved algorithms have yet to influence the existing reliance on diff.

2.3.3.2. Code based

From a maintenance viewpoint, understanding the differences is more important than producing the smallest possible delta. Horwitz [Horwitz90] argues that

> *"[Using diff to compare programs] can be unsatisfactory because no distinction can be made between textual and semantic changes."*

For example, diff will record an extra space as a difference, whereas if this space was being used for indentation, it will make no difference to the behaviour of the program. Alternatively, if block commenting is used, a program line may not show up as a difference, even though it may be commented in one version and not in another.

To remedy these problems, a number of syntactic based tools have been proposed. Horwitz [Horwitz90] proposes a technique relying on a simple language - one without procedures, pointers or arrays. The technique attempts to classify components in the new program that have been changed from the old, either as a textual or a semantic change. A semantic change is one where a component

exists in the new program without there being a corresponding component in the old, or alternatively a different behaviour is observed by the new component - either a different assignment sequence, evaluation sequence or output sequence. A textual change indicates a difference not affecting the semantics of the program. The technique relies on building a Program Representation Graph (PRG), combining control and data dependence. This is then partitioned so that any components with the same behaviour are contained within the same partition [Yang89]. Finally, changes can be classified as textual or semantic through the use of editor tags, recording when a change was made. Alternatively, a matching algorithm can be applied to the graph, although this is an expensive solution.

Binkley [Binkley96] extended this approach using program slicing to solve the problem of identifying semantic differences between the old and new programs. A solution to the identification of components with different behaviour is given, together with one that will correctly merge two programs together. As with Horwitz, the first stage of the algorithm requires a PRG to be generated. This is done for efficiency, and results in a linear-time operation. Data-flow analysis could be done instead, which would determine which statements create a variable, or use that variable before redefinition, although this would be far less efficient. Having achieved this, it is necessary to compare backward slices of the vertices in the dependency graphs of the old and new programs. If, given the same input, vertices result in two slices that have the same output, then they have the same behaviour and therefore any differences may be attributed to textual changes. If this is not the case, the differences indicate that the component has been changed semantically, or is a new component.

Yang [Yang91] takes a different approach, based on the fact that a programming language has a rigid syntactic structure, and that the comparison tool should therefore exploit this. The idea is that a comparison should be made on the parse trees of the source code, rather than simply on a line by line basis. This is a two-stage process. Firstly, the tool parses the programs into a parse-tree variant. A node represents a token or a sub-structure. A node appearing in one tree that is absent in the second indicates an addition or deletion to the code. If the contents of the two nodes are different, this indicates the code has been changed. No semantic checking takes place to allow the tool to work on incomplete programs. The parse tree also includes comments and pre-processor commands. A tree-matching algorithm is then applied to the two parse trees, which may be configured to give certain emphasis to different types of match, such as matching method names or parameter types. The tool is about five times slower than diff, but Yang claims that 'the output is more accurate and easier to understand'. In addition, various heuristics may be applied to the algorithm in order to reduce the time further, such as only attempting to match identical function names. The developed tool works with programs developed in C.

Of these three, only Yang suggests an implementable solution to the problem of precisely identifying differences in source code. Although Horwitz and Binkley have a more comprehensive solution to the problem, in practice the available language (no pointers, global variables or multiple argument procedures) is too limited to be feasible for a system of any reasonable size and structure.

2.3.3.3. Clone Detection

The presence and problems of clones within software systems is well documented [Burd97]. Clones are usually formed when a maintainer needs to implement new functionality, although there are some alternative reasons [Baxter98]. The code will be examined in order to find similar functionality elsewhere in the system. If this is found, one of two approaches will be taken. Either the maintainer will understand the code, but not wish to parameterise it in order to provide the added functionality due to the danger of ripple effects, or will not understand it fully and so copy the whole code into the new location. Once copied, small changes such as renaming functions or variable names may occur. Also, a small amount of code may be added in order to achieve the new functionality desired.

The danger of using clones in this way is twofold. Firstly, the code will increase in size rapidly when implementing new functionality, as the modification is much larger than necessary. As more code exists, the cost to maintain the code will increase. Additionally, the presence of the same code in more than one location will mean that any existing bugs will have to be fixed in every location. Over time, as clones are forgotten, this will become more difficult. The other problem is that if the code is copied without the maintainer having comprehended the behaviour, it is probable that unnecessary, unexecuted code would also be copied. This will also make future maintenance more difficult, as maintainers spend time determining the impact of the dead code.

Clone detection is a very similar problem to that of identifying differences, or change tracking. Whereas change tracking seeks to identify all the changes that occur within two versions, clone detection aims to spot chunks of identical code within the same system. Thus change tracking can be seen as clone detection across two versions, with changes being indicated by an absence of clones. Clone detection is also relevant for determining the origin of any new code generated. If cloning was used to generate new code in the most recent version then this could be highlighted specifically.

A number of clone detection systems have been developed which will be investigated. As the benefits of clone detection are easily quantifiable, the problem has received much more recent research interest than change tracking. The important difference to consider is that clone detection systems need to avoid false positives – the reporting of a clone where one does not exist – as this makes them unsuitable for future possible automation. Therefore, the number of clones reported is likely to be conservative. Although adequate for clone detection, if change tracking is seen as the inverse result, then the number of changes reported will be excessive.

Three basic approaches to the problem exist, working at different granularities. Johnson [Johnson94] aims to identify clones by matching common substrings across the system. The source code is initially modified in a number of ways to reduce the impact of alternative layouts. This may include removing whitespace within the code. Alternatively, comments may be removed or used exclusively. Finally, a marker may be used instead of an identifier to resolve changes due to renaming. Substrings are then generated, ensuring every character within the code appears in at least one substring. These are then fingerprinted, so that similar substrings have similar fingerprints, and different substrings have

different fingerprints. This allows the comparison routine to be far more efficient. Finally, substrings are matched, with the emphasis on finding the longest substrings available.

Baker [BBaker95] also takes a text-based approach to the problem. Here, a more flexible approach is taken, with better support for renaming of variables within the cloned text. This is done by an approach referred to as parameterised matching. The process involves replacing identifiers within the source code with parameter symbols, and the source is then encoded. The encoding measures the distance in the string since the previous occurrence of that symbol, excluding white space. For example, $x = x + y$ is encoded as $p = p + p$; x, x, y and then $0 = 2 + 0$. The line $a = a + b$ would result in the same encoding, and so the two lines are considered to be clones. In practice, the author states that the approach works best on clones of 20 lines or more. The technique works in approximately linear time given the input length, although with a quadratic worse-case scenario.

Mayrand et al. [Mayrand96] take an alternative approach to the problem. Their system analyses clones at the function level, by comparing 21 metrics based on the code. Four criteria are used for identification – function names, code layout, expressions and control flow, with the latter three determined by metrics. The layout metrics are based on the number of comments, lines of code, and length of variables. The expression metrics include measurements such as the number of calls to other functions, the code complexity and the number of declaration and executable statements amongst other features. Finally, the control flow metrics consider features such as the number of available decisions possible, the nesting level within the code, and the presence of loops.

If two sections of code have identical metrics, then they may be considered to be clones. However, the technique also allows weaker matches to be identified by reducing the number of conditions that must be matched, such as by disregarding the function name or the length of the two sections. Finally, a poor control flow match, once the other categories have been disregarded, shows that the two sections of code are distinct. The system is currently less efficient than the text-based approach, with polynomial runtime. Also, the technique relies on the accuracy of the metric tool (in this case Datrix [Datrix03]) such as how complete it is, and whether pre-processor statements are resolved.

Finally, Baxter et al [Baxter98] based a clone detection tool upon abstract syntax trees. One of the benefits of this, they argue, is that clones can be removed from the source using standard transformations. The algorithm works by comparing sub trees within the abstract syntax tree (AST), in a similar way to Yang [Yang91]. The problems of scale and similar (near-miss) clones are addressed however, and could also be applied to Yang's approach. The approach works by fingerprinting sub-trees using a hashing function, so sub-trees that are identical are given the same hash code. This fingerprinting allows the operation to take linear time, dependent on the number of nodes within the AST. This resolves the scale issue. By using a less precise hashing function, near-miss clones can also be identified. Clones may be classified according to how similar the underlying sub trees are, by considering the number of shared nodes compared to the number of different nodes. The algorithm also resolves some of the problems that arise as a result of using an AST, such as reordering commutative operations in order to detect more matches.

Each of the approaches considered has some problems. Text-based solutions cannot cope with changes more complex than renaming, such as replacing an identifier with a function call, or handling commutativity and similar trivial transformations. Code-based solutions incur the additional overhead of parsing the code and managing any errors produced if the code is incomplete. This parsing may also introduce difficulties with pre-processor macros. Apart from the metrics approach, they all also have problems with the irrelevant interchanging of lines within a potential clone.

However, the abstract syntax tree approach returns the best results for change tracking. Therefore, given the necessary overhead in order to generate the tree, the useful addition of clone detection can be achieved almost for free. Therefore, this would currently appear to be the best solution for change comprehension purposes.

2.3.4. Summary

This section has summarised some of the existing research in software configuration management. Software configuration management is an important aspect of managing software evolution, and particularly that associated with open source development. The standard tool for open source use is CVS, even though it has a number of problems that have been resolved by other configuration management systems. Furthermore, CVS is likely to remain the standard tool for the near future.

The second part of this section addressed the means by which differences between two versions may be identified. Although this is an important part of a configuration management system in order to reduce the storage requirements, the ability to view differences is also useful from a maintenance perspective. A number of different techniques were summarised, of which a syntax based approach offers the best compromise between accuracy and flexibility.

2.4. Software Visualisation

Card et al. [Card99] define information visualisation as:

> *"The use of computer-supported, interactive, visual representations of abstract data to amplify cognition."*

Software visualisation is generally considered to be a specific branch of information visualisation, treating software as the 'abstract data'. To convey this idea, Stasko [Stasko00] expands Card et al's statement, saying that:

> *"Software visualization is the use of computer graphics and animation to help illustrate and present computer programs, processes and algorithms."*

Knight and Munro [Knight99] give a more general definition, emphasising the need for intelligence amplification [Knight00b] within visualisations.

"Software visualisation aims to aid the programmer by providing insight and understanding through graphical displays and views, and to reduce the perceived complexity through the use of suitable abstractions and metaphors."

This section will firstly examine the motivation for using software visualisation from a cognitive perspective, and list several guidelines that should be considered when developing visualisations. Secondly, a number of generic techniques that are used within information visualisations to display different forms of data will be summarised. Thirdly, the features found within software visualisation systems, and the difficulties of providing these features will be addressed. Finally, a number of existing visualisations that highlight evolution in some manner will be evaluated.

2.4.1. Cognitive issues

2.4.1.1. Why software visualisation is beneficial

Card et al. [Card99] suggest that there are several benefits of using an information visualisation system. These are summarised as follows:

1. Further resources are provided to the user. By visually displaying some of the user's understanding of the data, the user does not have to store this information mentally, and so allowing that memory to be freed for other tasks. Additionally, some operations allow the user to work with perceptual abilities. For example, the visualisation may exploit Gestalt properties, such as the perceiving of grouped data to be related.

2. The time taken to search for information is reduced. This may be achieved by increasing the density of data displayed, or grouping related data.

3. Patterns and trends in the data may be spotted more easily. Spotting patterns visually is much easier than recalling that information from memory and processing it mentally. Also, through filtering and abstraction methods, visualisations can present a manageable amount of relevant information rather than an entire data set.

4. A visualisation allows monitoring of several states simultaneously, particularly if the display is organised so that these states are highlighted.

5. A visualisation has some benefits over a static diagram, because it allows the user to explore the environment, requesting further details or filtering the data as necessary.

As the previous definitions reveal, software visualisation is a specialisation of information visualisation, and so the above benefits should also apply to software visualisations. Further motivation for creating software visualisation systems comes from Myers [Myers90], who suggests that using graphics to present software can remove the focus away from the syntactic issues inherent within code, and instead provide a higher level of abstraction. In addition, complex programs such as concurrent or real-time systems that are difficult to follow textually may be better represented graphically.

2.4.1.2. 'Good' visualisations

Given the potential benefits of a visualisation, it is important to be able to maximise these benefits when designing the visualisation. In general, the knowledge about the cognitive value of graphical representations is limited [Scaife96]. However, there is some research into the features of a good visualisation, from cognitive and usability viewpoints.

Larkin and Simon [Larkin87] addressed the issue of diagrams and their usefulness. In particular, they concentrated on diagrams such as those that might be used by a physicist in order to solve a problem. They concluded that diagrams have three useful features. Firstly, diagrams can group together all the information that is needed to solve the problem, rather than requiring a large amount of searching to find the important elements. Related information about an element may be located near by, and so the need to match symbolic labels over a large distance is reduced. Finally, a large number of perceptual inferences are supported automatically.

They argued that to be useful a diagram must be constructed in such a way as to take advantage of these features. Failing to do so is probably part of the reason why diagrams may sometimes seem to be ineffective. Diagrams were seen to be advantageous not just because a diagram contains more information than the equivalent wordy problem, but also because the indexing of this information can support extremely useful and efficient computational processes.

Scaife and Rogers [Scaife96] concluded that both the organisation of information on a display and the notation used for representation are crucial. They recommend allowing opportunities for external manipulation of the diagram, for example, by adding annotations. Multiple representations are also suggested, although they recognise that this may raise further issues. Various other features are listed as follows.

Explicitness and visibility. By directing attention to the key components which are important for the task, the process of inferring information from the diagram is simplified. Highlighting patterns is also important.

Cognitive tracing. When using static diagrams it is possible to allow the user to mark and highlight information easily, although the support for interaction is low. If virtual reality or animation is being used then marking information in an effective manner becomes harder.

Ease of production. The authors argue that "a history of being taught to draw diagrams makes for fewer problems with understanding new ones". This means the diagram should be easy to produce or modify, thus allowing the user to become familiar with it.

Combining external representations. In some cases, text is necessary for understanding a diagram, whether through labels, or a key, or some other method. However, separating text and diagram "increases the computational load involved in comprehension". This is more significant when animation or virtual environments are involved. Narration was seen as a more successful approach, and harder to ignore than text-based labels.

Distributed graphical representations. Diagrams offer the possibility for many people to add elements to the same diagram simultaneously. It is possible for additions to be recorded, and later animated to show how the diagram evolved. Similarly, virtual environments allow constructions of graphical representations for users in dispersed locations.

Petre et al. [Petre98] question whether software visualisation is really the powerful cognitive tool that it is claimed to be. This question is answered by examining six issues, which provide further evidence of what constitutes a good visualisation.

Suitability refers to the relevance of using a visualisation in the first place. They conclude that software visualisation is worthwhile under any one of three conditions. Firstly, if there is the need to analyse a large data set and identify trends and patterns. Secondly, if the execution of a program is unclear. Finally, if unknown relationships within the data need to be identified, which may be revealed by the provision of alternative viewpoints.

The role of the visualisation must be addressed, for both novice and expert users. They introduce the idea of a visualisation being an important communication tool for experts, ensuring that everyone has the same understanding. Novice users can also benefit from this by gaining an insight into how experts operate.

They also observed that experts were likely to use visualisation tools for taking over a program they are unfamiliar with, particularly if it is large or requires multiple technologies. Therefore the tools had to cope with large amounts of data. The experts also wanted accurate visualisations, preferring speed to beauty, and requested control over the visualisation.

Although the authors recommend only one view, they realise that the amount of data involved is unlikely to make this possible. They suggest that switching between representations depending on the current task may be appropriate. Alternatively, a view providing a different viewpoint, such as providing a 2D overview with a 3D terrain may aid understanding. They also suggest a concept of "useful awkwardness" – that is, by forcing the user to switch representations, they may notice unusual features or discontinuities between the two views that they would not otherwise have observed.

Finally, the role of graphics as opposed to text was investigated, by interviewing programmers. Graphical representations were described as 'more fun', 'more comprehensible' and 'easier to understand'. Further questioning showed that the actual benefits were less clear. However, the authors conclude that the illusion of understanding may be more important than the reality, and that satisfaction with the tool may offset any overheads caused through the use of graphics.

Shneiderman [Shneiderman96] details a series of general tasks that a visualisation should be able to support. These are summarised by the 'Visual Information Seeking Mantra' - "overview first, zoom and filter, then details on demand". More specifically, the necessary tasks are:

Overview: A visualisation should present a complete picture of the system, with support for analysing individual areas in more detail.

Zoom: The ability to zoom in on one area in more detail, with control over both the magnitude and focus.

Filter: A user should be able to reduce the amount of information on display by hiding or eliminating unwanted items. This facility can not be provided by using navigation and zooming techniques alone [Carr99].

Details: A user should be able to request further details of items, or collections of items, within the visualisation. This can be achieved through the use of pop-up windows.

Relate: The visualisation should allow relationships between items within the visualisations to be viewed. Shneiderman considers this the most important aspect of visualisations, particularly when analysing complex and interconnected data.

History: A list of actions used within the visualisation should be stored, to allow undo and redo operations. This also allows operations to be combined or replayed for future reference.

Extract: Once interesting data has been identified within the visualisation, support should exist for extracting this to allow further processing. This could allow dedicated statistical analysis, for example.

Unfortunately, it is almost impossible to include all of these features within a single visualisation. For example, the scale of the visualisation required by experts, with the large number of complex relationships, may mean that there is not the space to display related, labelled, items together. Therefore, compromises must be made with an emphasis on certain items or relationships. There are a number of existing techniques that may be used in order to do this.

2.4.2. Information visualisation techniques

Representation is an important aspect of any information visualisation system. This concerns the mapping of data values to some graphical display. A good representation should take into account the features recommended within the previous section. However, it will also depend on the structure of the data, and the purpose of the visualisation itself. Some novel representations will now be presented.

2.4.2.1. Distortion techniques

Distortion techniques aim to present the data in such a way as to focus on the important data, or the data the user is interested in, whilst still displaying the surrounding data.

This type of view simulates the effect of placing a wide-angle lens (fish-eye) over the image. The visual effect is that objects in the centre of the image are greatly magnified, whereas those at the edge are reduced in size. This is often a very valuable technique, as it allows an interesting object to be viewed in some detail, whilst simultaneously maintaining the context of this object within the image. Furnas [Furnas86] originally proposed the technique, with later work by Sarkar tailoring this for use with graphs [Sarkar94], and Robertson and Mackinlay applying it to documents with the Document Lens [Robertson93].

This idea was developed further into the 'Table Lens' [Rao94]. The system is designed to visualise and comprehend large tables such as a spreadsheet. The lens is designed to maintain coherence of rows and columns, which would be bent and distorted using the standard technique. The result is that the table maintains a rigid grid structure, with rows further from the cell under consideration reduced in height proportional to the distance, and columns further from the cell similarly reduced in width. The other main difference from a traditional fisheye view is the support provided for multiple focal points.

An alternative distortion technique can arise through the use of perspective in a 3D environment. The Perspective Wall [Mackinlay91] is a technique for viewing and navigating linear information. As with other distortion techniques, the aim is to create a view that supports both context and detail. The effect is similar to that of the table lens, with the width of rows remaining constant but columns reducing in size in proportion to the distance from the focal point. Navigation is achieved by rotating the wall to bring the next section into focus. The main problem with this approach is that due to the 3D environment, large areas of the screen are unused in order to emphasise the perspective view.

2.4.2.2. Pixel Oriented Techniques

The basic idea of a pixel oriented technique is to map each data value to a pixel or glyph, where the colour of the pixel is determined by the value of the data. The problem then becomes one of arranging these pixels meaningfully. Different arrangements are useful for different purposes. Keim et al. [Keim96] separate these into two types.

A **query-independent** technique is useful when the data has a natural ordering, such as time. A simple arrangement involves placing the data from left to right in a line by line fashion. However, Keim et al. suggest that in general this does not produce a meaningful result. Instead, techniques that cluster similar data items are more useful, such as a recursive pattern technique [Keim95]. The idea is that associated data is grouped into rectangular blocks, and it is these blocks that are then used to create the overall pattern. For example, a block could contain data recorded on a particular day. Seven of these blocks could be grouped again to show the data over one week, and so on. Investigations suggested that the most appropriate layout for the blocks was in a back and forth fashion, from left to right and then right to left.

A **query dependent** technique should be used when the data has no natural ordering, or when exploring other relationships in the data. In this case, the distance from the actual data values to the values specified in a query determines the order that is used when arranging the data. The importance of a particular dimension, or dimensions, can be considered by applying weightings to each dimension. An example of this is the spiral technique [Keim94]. Each item of data is placed into a block (as square as possible) with $n+1$ segments, where n is the number of data attributes. The first segment is coloured according to the closeness of the data block to the query. The remaining segments are coloured by mapping the values of the attributes to some colour scale. The blocks are then ordered in order of closeness, and placed in a square spiral starting from the centre. Patterns may be seen easily as blocks the same distance away from the centre have a similar closeness to the query. However, using a square

spiral to conserve screen space means that navigating the spiral is more difficult than if it was a gradual curve. Circle Segments [Ankherst96] is a similar technique, where a circle is split up into *n* sectors. Each sector represents one attribute of the data. The data is ordered according to closeness, and then each sector is filled by mapping the data values to coloured pixels. The main problem is that as the number of data items increases, the size of the circle also increases, making comparisons between two attributes on opposite sides of the circle more difficult. Rearranging the order of attributes goes some way to solving this problem. Also, it is easy to become lost comparing items in the centre of a sector, although highlighting the same data item in other sectors would solve this problem.

An alternative example of these techniques is the starfield display [Jog95]. Items from a database are plotted in two-dimensional space as small, coloured, selectable glyphs, using two of the ordinal attributes of the data as the variables along the display axes. Approximately one million data items may be displayed simultaneously in real time, which is critical if the visualisation is to be interactive. This is particularly important as it allows real-time zooming and panning within the visualisation. Two examples of this display are the query independent Dynamic Homefinder system [Ahlberg94], where stars are plotted according to their location to show the properties of houses satisfying various properties, and the query dependent FilmFinder system [Jog95], where stars reflect the year of release and popularity of a film.

2.4.2.3. Hierarchical Techniques

Hierarchical techniques require the data to have a hierarchical structure. This may be inherent, or it may be possible to force the data into such a structure. The Tree-Map technique [Johnson91] aims to show this structure in a two dimensional display, using all of the display space available. A Tree-Map partitions the display space into a collection of rectangular bounding boxes representing the tree structure. The size of the bounding box is proportional to the weight of a node in the hierarchy. Any additive metric may be used for the weight - i.e. the weight of a node must be greater than, or equal to the sum of the weights of its children. The bounding boxes of any children of a node are placed inside the bounding box of that node, with a ninety-degree rotation. Various properties, such as colour and texture can be applied to the rectangle drawn inside the bounding box, in order to display further details about the element. The result is a view that, with practice, shows the position and value of a node within a hierarchical structure. However, nodes may not be visible if the weight assigned is small compared to other nodes. This problem is amplified for nodes with a high depth. Zooming is suggested to solve this problem, but as the location of a node in a hierarchy can only be determined by viewing neighbouring nodes, it is likely that some distortion technique would have to be applied.

Cone Trees [Robertson91] are used to display a hierarchical structure using three dimensions. The structure is placed in a 'room', with the root of the hierarchy placed near the ceiling, at the apex of a transparent cone. The children of this node are placed along the edge of the base of the cone, equidistantly spaced. The next layer of the hierarchy is built up similarly, resulting in a hierarchy of cones. The height of the cones is constant throughout the representation, whereas the radius of the cone base is reduced at each level, allowing the leaf nodes at the lowest level to still be displayed within the

room. When a node is selected, the cones rotate in parallel so that the route from the selected node to the root of the tree is highlighted. Alternatively, the tree can be continuously rotated to demonstrate the hierarchy. The rotation is animated smoothly to maintain context. The main problem with this approach is that the 3D display obscures labels associated with the nodes. However, the nodes themselves are semi-transparent which alleviates this problem to some extent.

The Information Cube [Rekimoto93] visualises a hierarchical structure as a series of nested cubes. The outermost cube corresponds to the root of the structure, and contains cubes representing the children of the root. Each of these cubes contains further cubes, and the process continues until the leaf nodes are reached. These are represented as a tile within the parent cube, rather than a further cube. Cubes are labelled on the outer faces to show the node name.

The cubes and labels are rendered with partial transparently, allowing the children within a cube to be clearly visible whilst still allowing the node to be viewed. This has the result that leaf nodes may be easily traced back up through the hierarchy. Transparency was found to be significantly more useful than a wire-frame representation for this reason, as a wire-frame view appears to create a more complex image. The authors also suggest that the transparent surfaces can convey other information about a node, such as the value of a metric, by changing the opaqueness of the surface. Nodes can be viewed in more detail by zooming using animation, and the structure may be rotated around this point.

The Information Cube has many advantages over a cone-tree when the hierarchical structure is wide and shallow, as less overlapping nodes are created. However, as the tree becomes deeper, it is more difficult to distinguish between nodes further down the hierarchy without zooming in, as the cubes decrease in size and are likely to have labels of cubes further up the hierarchy obscuring them. Also, it is not clear how many transparent surfaces can be placed together before the surface becomes practically opaque.

2.4.2.4. Hybrid Techniques

Finally, these techniques may be combined to produce other systems. For example, Andrews [Andrews95] implements a three-dimensional information landscape within a larger visualisation system. The landscape is generated by mapping collections of documents onto a plane in the form of towers. Collections are hierarchical, with the possibility of many documents belonging to a collection. Colour is used within the visualisation to represent the type of document being viewed, with the height of the tower indicating the size of the represented documents. A two-dimensional map is also provided, acting as a birds-eye view of the plane. In addition, consideration is given to using three-dimensional models within the landscape to represent the contents of a document. For example, using a model of the Eiffel tower would represent the fact that the underlying documents are related to Paris. This has the additional benefit of acting as a navigational aid to prevent the user becoming disoriented within the landscape.

2.4.3. Software visualisation

2.4.3.1. Summary of taxonomies

Over the last decade, several taxonomies have been produced in order to try and classify the software visualisation systems in existence at the time.

Roman and Cox [Roman93] define five separate dimensions for classification, with a heavy emphasis on algorithm animation systems. Together with abstraction and presentation, scope, content and interaction were all considered. Price [Price93] introduced a similar taxonomy, which contains more detail about these latter aspects, along with form, method and effectiveness. Together, these are summarised as follows.

Scope is the range of programs the visualisation system can handle, and considers scalability as well as the programming languages that the system supports.

Content considers the information visualised, whether source code or algorithms. Consideration is also given to the completeness of the visualisation, and the time required to generate it.

Form takes into account the characteristics of the output of the system. This includes issues such as the target medium - paper or monitor, levels of granularity, the presentation forms available such as colour, animation or sound, and the ability to have multiple views or multiple programs.

Abstraction indicates the type and detail of information that is conveyed, and is required to control complexity and increase understanding. Many visualisation systems will support more than one abstraction level simultaneously and allow inclusion of further implicit or derived information.

Presentation considers how the system conveys information. This includes the interpretation given to graphics, where shapes or patterns suggest an underlying meaning such as a relationship, or a ring shape to represent a repeating sequence. An analytical presentation focuses on displaying information useful for further analysis of the system, such as metrics or correctness. An explanatory presentation may be useful to demonstrate particular areas of an algorithm, focusing in some detail on a difficult concept, for example. Finally, an orchestration may display many visualisations of slightly different algorithms solving the same problem, such as a display of multiple sort algorithms.

Method categorises systems on the method required to create the visualisation, such as whether it must be programmed explicitly or generated automatically, and the level of potential customisation. A further factor is to consider how the software and visualisation are linked, in order to determine whether it is necessary to modify the software in order to create the visualisation.

Interaction considers both navigation and control. Style covers the various methods of inputting instructions into the system, whether through a GUI or command line. Navigation covers issues of the ability to cull information, and also of temporal control, such as the speed or direction of an algorithm animation. Finally, the extent to which scripting or macros are supported is considered.

Effectiveness tries to capture how well a system communicates information to the user of the visualisation. This requires the purpose of the visualisation to be well defined, and evaluated empirically. Consideration is also given to the length of time that the system has been used.

An alternative taxonomy, with a greater focus on the task required by the user of the visualisation, was proposed by Maletic et al [Maletic02]. This taxonomy includes many of the criteria proposed by Price [Price93], although some have been expanded to reflect the advancements which have been made in the field of software visualisation. The categories are ordered from most to least important, from the perspective of a user.

Task – why the visualisation is needed. This specifies which software engineering task, or tasks, are supported by the system. These can include education, programming, debugging, faultfinding, and process management. The task is likely to influence the data structures and views provided by the system.

Audience – who will use the visualisation. Many algorithm animation systems are aimed at students within an educational setting. Automatic debugging tools are more likely to be aimed at experienced developers. Training is considered to be another important aspect, as some tools are more difficult to use than others, and require a greater initial investment of time. A simple tool will often result in limited functionality, restricting the audience.

Target – the data source to be represented. This defines the low-level aspects of the software to be visualised, such as the source code, metric data, design documentation, or the results of running test suites. Scalability is also an important aspect, as this will influence the available representations and media.

Representation – how to present the data. This is driven by information visualisation and cognitive research. Views are created by mapping the underlying data to a chosen metaphor or representation. This representation must consider two aspects – expressiveness and effectiveness. Expressiveness is an indication of the completeness of the metaphor, for example whether the metaphor allows the representation of all the attributes that should be shown. Effectiveness determines the usability of the metaphor, such as whether the important information within the data may be seen easily.

Medium – where to display the visualisation. Traditional media include paper and ink, and small monitors. Paper requires static non-interactive visualisations, whereas monitors are slightly more flexible. New technology is providing alternatives however such as room-sized displays, stereoscopic screens, or virtual reality environments. These new developments allow alternative representations with some of the scale and 3D navigational issues removed.

2.4.3.2. Challenges in software visualisation

There are many challenges facing the developer of a software visualisation system. Together with the ideal features of hypothetical visualisations presented earlier, there are also a number of other aspects involved. Many of these result from the need for automation and the desire to present large data sets.

This is because as the amount of data to be visualised increases, developing handcrafted visualisations becomes increasingly less practical and cost-effective, and there is a far higher probability of making errors.

Scale is a key part of software visualisation, and has a significant influence on appropriate representations [Maletic02]. Eick and Karr [Eick00] list six factors that affect the scalability of a visualisation. Fundamentally, human perception is the most significant, as everything depends on how effectively the human eye and brain can interact with and process the display. Following on from this the actual representation used, such as bar charts or cityscapes, and the interaction options available, are also important. Finally, from a technical viewpoint, the scalability of the algorithms and data structures used within the system, the size and resolution of the display, and the available storage space and processor power may also have some effect. They also present a number of enhancements that can be made to current simple representations to improve the scalability, using abstraction techniques to increase the amount of data displayed. Many of these are based on a combination of zooming support with a level-of-detail approach, which they refer to as a 'multi-resolution metaphor'. For example, zooming out on a scatterplot may group nearby individual items into one larger group, indicated this through the use of colour. The information mural [Stasko96] is an example of a similar idea, applied to line graphs. Different algorithms are also used to take account of the larger data set, for example, by plotting items in a different order to ensure the most important values are displayed last and not rendering items that would be displayed behind them.

Developing suitable layout algorithms for the elements within information and software visualisations is a further difficult problem to solve. A correct layout of items will allow patterns in the data to be spotted far more easily than a poor layout. There are few concrete rules of what constitutes a good layout, although Purchase et al [Purchase02] have identified some guidelines for graph-based representations. However, these have only been verified for graphs of small scales. In general, layout algorithms are a continual balancing act between processing power, aesthetics, and perceptual inferences. For example, an otherwise good graph layout may result in a cluster of unrelated nodes, which are grouped perceptually by the user. The already difficult problem is extended when evolving or dynamic data is introduced to the visualisation. At least three graph layout algorithms exist that handle this situation better than most. A spring-graph algorithm [Battista99] is flexible enough to allow changes to the structure, although some changes may produce a radically different layout. Gnutellavision [Yee01] uses a radial layout based on NicheWorks [Wills99] together with animation to produce a reasonably successful evolving graph layout. Finally, DynaDAG [North96] concentrates on producing layouts for evolving trees, such as for inheritance hierarchies.

Knight [Knight00a] investigated two approaches to solving some layout problems in a more general visualisation setting. The first of these involved an initial space allocation larger than that required, in order to allow future expansion. The second involved an abstraction mechanism appropriate for 3D environments referred to as world-within-world, where worlds can be manipulated without impacting upon others. In general however, there has been very little research into the difficulties and pitfalls of incorporating evolution within visualisations, and this will be addressed further in chapter 3.

All visualisations will also require some form of mapping between data and picture. In some cases, such as Software World [Knight00a], this mapping may take the form of a real-world metaphor. In this example, cities represent classes within the code, and buildings represent methods within those classes. The main purpose of a real-world metaphor is to ease the cognitive burden on the user. According to Benford et al. [Benf96] the use of such metaphors is:

> *" an attempt to exploit people's natural understanding of the physical world, including spatial factors in perception and navigation, as well as general familiarity with common spatial environments."*

This should be achieved by maintaining a close relationship between data and metaphor. For example, data of a hierarchical nature is best represented using a hierarchical metaphor. Also, important items within the data, such as a long file with a high complexity metric for example, should be represented in an uncommon fashion within the metaphor - such as by having an unusually high, badly built building. The opposite is also true - any unusual real-world representations within the environment, which are therefore likely to attract attention, should reflect unusual data. This is less difficult to achieve when using an abstract metaphor, as more flexibility is allowed within the environment. However, the benefit of the perceived lower cognitive overhead, and the alert of an unusual state within the environment, is also lost. As with layout, there are a number of issues regarding the use of evolutionary metaphors within visualisation systems that have not been considered.

2.4.4. Evolutionary visualisations

In the context of this thesis, evolutionary visualisations are those which are able to visualise time-based data. This section summarises a number of these visualisations.

2.4.4.1. Information visualisations

Time Tubes [Chi98] are designed to show the evolution of the structure of a web site, although they could also be used to show the evolution of any hierarchical structure. A Time Tube is composed of one or more Disk Trees, with each disk tree displaying a snapshot of the web site at a given moment in time. A Disk Tree is similar to a cone tree, except that the structure is represented compactly in two dimensions. In addition, rather than children being spaced equally around the base of the cone, spacing is carefully managed to ensure that a greater proportion of space is allocated to large subtrees. The effect of this is that overlapping of nodes is prevented, thus allowing every node within the hierarchy to be viewed simultaneously without requiring zooming or rotation. Nodes (web pages) and edges (links) are coloured, representing the number of accesses.

Two methods are suggested for the layout of disk trees. The first is that disk trees should be laid out along the horizontal axis without overlap, where the horizontal value corresponds to the time period that the disk tree represents. Therefore, comparing a disk tree to its neighbour allows the user to view the modifications that have taken place to the structure at the two different times. Disk Trees may also be rotated around a vertical axis to allow more Disk Trees to be placed along the horizontal axis. The

degree of rotation allows further variables to be mapped, although this reduces readability. The second option is that only one Disk Tree is displayed at once, and animation is used to display these in chronological order. This allows the Disk Trees to be larger than when many are displayed simultaneously, and the ability to identify immediate changes is enhanced. However, identifying changes that occur over a large amount of time is significantly more difficult.

In order to maintain consistency, and to reduce the cognitive load during animation, every node that existed at some point during the time period is shown on every disk tree. This means that nodes remain in the same place throughout. This is particularly important in a representation such as the Disk Tree where nodes can move around significantly depending on other, possibly distant, changes to the hierarchy. However, no consideration has been given to the long-term consistency of the visualisation. For example, although a disk tree remains constant during the time period intended, extending this time period may mean that a completely different shape of disk tree is created. Therefore, any knowledge of patterns gained from viewing the first visualisation may be difficult to recover when presented with the second.

A Spiral Graph [Weber01] is intended to show periodic behaviour in time-based data. A pixel-oriented technique is used, with each data value mapped onto a section of the spiral. Each ring of the spiral represents the same period of time, and so the length of each section increases as the spiral becomes larger. An example highlighting the sunlight intensity measured over a period of a month is shown in Figure 2-1.

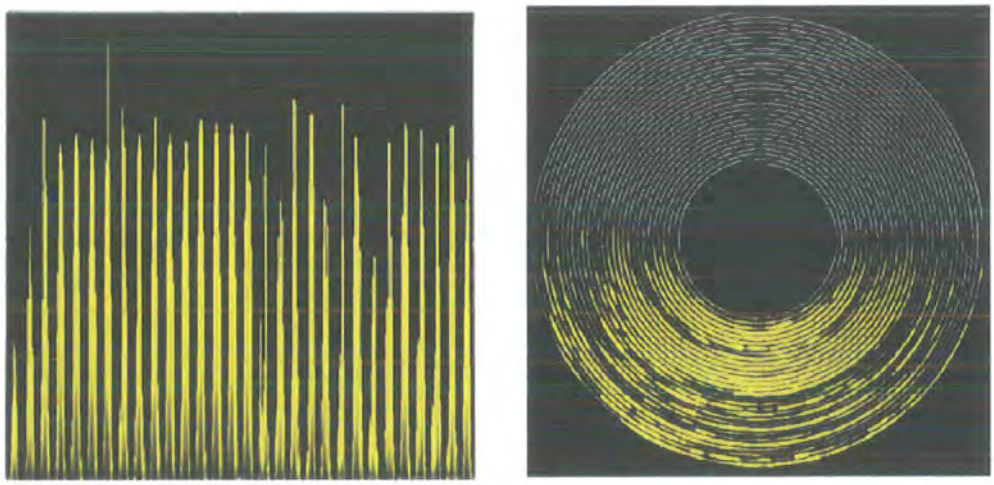


Figure 2-1. Left, a chart of sunshine intensity. Right, the same data as a spiral graph.

If the time period is not known, then a default period is initially used. The spiral is then animated, with the spiral redrawn each frame with a continuously increasing time period. The intention is that the user initially perceives the display as unstructured. As the time period used within the visualisation nears that contained within the data, the display becomes more structured, until the time period is passed and the display becomes unstructured again. Early experiments showed this was a very effective approach.

The visualisation was then extended to allow multiple spirals in order to allow several data sets, or attributes of the same data set, to be compared simultaneously. Additionally, the visualisation does not suffer from issues of consistency, as the latest data may be placed on the outer edges of the spiral without affecting the remainder of the display. The main drawback of the visualisation is that although it highlights time-based relationships, it is difficult to view any other relationships contained within the data.

2.4.4.2. Low level software visualisations

Software evolution can be viewed at a number of levels of granularity. Existing text based tools, such as diff, focus on highlighting whether individual lines of code have changed between two versions. This section summarises a number of visualisations of similar low-level evolution.

2.4.4.2.1. SeeSoft

SeeSoft [Eick92, Ball96] was designed to produce a method of understanding statistics collected at the source-code level of detail. The problem was how to display the millions of lines of code in a meaningful manner. No known visualisation methods were found useful, and so a new technique was developed based on four key ideas – reduced representation, colour, direct manipulation, and access to the underlying code.

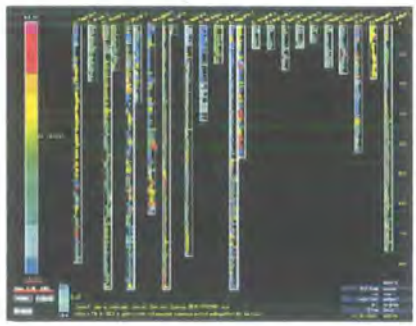
Reduced representation is achieved by representing files in the program to be visualised using columns, and lines of code within those files by using rows of colour. The length and indentation of each row can correlate to the actual line of source code, so the display mimics the actual program. If there are too many rows in one column, the column is wrapped. The row colour used represents a metric or statistic, for example, the age of the code, or whether it calls a particular function. Two examples of this representation are shown in Figure 2-2.

Direct manipulation refers to the ability to select particular statistics using the mouse whereupon the display is updated in real time. Rather than clicking, a simple 'mouse over' operation is used, which the authors claim allows the viewing of several hundred statistics very quickly.

Finally, the ability to read the code is achieved by having a 'magnifying' box, where the actual code represented by a row is reproduced in a separate window using a mouse over operation.

On the surface, SeeSoft appears to be a very sensible and clever way of giving a brief overview of large amounts of code. The idea of correlating the appearance of the source code with the display is a good one, and should allow someone familiar with the source code to navigate quickly round the display. Similarly, the ability to view thousands of lines simultaneously is also a potentially very useful feature, particularly for detecting patterns spread across the code, as the authors suggest. Perhaps one of the best features though is the ability to use information from different versions of the same file, in order to view how the program has changed over time. By taking a SeeSoft column, and splitting it into a before and after view, far more information can be seen. This approach was taken by Burkwald et al.

[Burkwald98]. SeeSoft displays have also been used to solve other problems, such as predicting the location of faults within code [Jones01].



(1) Typical SeeSoft code view.

(2) View showing additional structure

Figure 2-2. Two views taken from SeeSoft.

2.4.4.2.2. SoftCities

SoftCities [Young99] aims to show the evolution of software using a virtual environment. Based on a city metaphor, a city represents a source code file, with buildings within that city representing the functions available. A city is represented as a disc, with buildings placed in a quadrant on the disc depending on the length and complexity of the code. Once placed, however, the location is fixed for future evolutions. This means that if a large building is seen in the section reserved for small buildings, then it is clear that this function has grown in length. Each city includes a spire, the colour of which is determined by the average complexity of the functions in the city. The colour and texture of buildings indicates the age of the underlying function code. Similarly, the colour of the disc represents the age of the file. Showing scaffolding in the later visualisation indicates changes to functions, whether growing or shrinking. Similarly, smog represents maintenance work within functions.

The approach taken means that only one software release may be seen at once. There is no direct support for viewing several releases, apart from by printing the results and making visual comparisons, which may turn into an expensive 'spot-the-difference' exercise [Scott00]. The nature of a 3D environment also makes this more difficult, particularly with large cities. However, comparison between a release and the next immediate release is clear and simple, with changes highlighted using both size and colour.

2.4.4.2.3. Software World

Software World [Knight99, 00b] expands the use of a city metaphor to visualise an entire software system developed in Java, although theoretically this should be adaptable to other modular or object oriented languages. This is achieved by providing visualisations at many different levels of abstraction, in order to handle the inevitable complexity. The levels include:

World - This represents the entire software system, and is visualised as an atlas-like picture.

Country - This maps to a directory structure, and is designed to break down the world into more manageable sections.

City - A city within a country represents the contents of a file, and is composed of many districts. In Software World, each district represents a class at the same scope level.

Buildings - A building within a district represents a method within that class. Buildings are laid out in a block, alphabetically. Each building has a different size, representing the number of lines of code in that method. In addition, parameters, variables, access rights and the method name are also displayed using various attributes of the building.

An example of the city view is shown in Figure 2-3. Two reasons are given for basing the city layout on a grid. Firstly, it prevents unwanted relationships being made from the placement of various buildings. Secondly, it allows the evolution of cities to be viewed more easily. As a building always remains in the same location, it enables the user to make comparisons between several versions of a method without having to search for the new location of the building for each version. Upon generation, the buildings will take up half of the available space within a block. This is to leave sufficient space for new buildings to be added to the city as the software evolves. The question of how evolution would be handled at the higher levels of abstraction is not discussed however.

Although a suitable metaphor for this task, particularly for a static snapshot of the system, there are other tasks required which would be more difficult to map cleanly to this visualisation. For example, no control or data flow is shown. Implementing this would require links between different cities, potentially in many different countries, and this may be difficult to achieve without losing the level by level abstraction chosen.

Figure 2-3. Image from Software World, showing two large methods and several smaller ones in a class.

2.4.4.2.4. Evolution Matrix

The evolution matrix [Lanza01] is a very simple approach for showing changes in classes over time. A class is represented as a rectangle, with one metric (the number of methods) mapped onto the width, and another (the number of instance variables) onto the height. These are then laid out on a grid, with the columns representing versions of the software, and rows showing the same class changing during those versions. New classes introduced are placed on new rows at the bottom of the grid. Classes removed will leave an empty space in the graph from then on. Three colours are used – black, grey and white, to show whether the classes grow, shrink, or stay the same size.

Although a simple technique, much can be seen from the view. Snapshots from different points in time can be seen easily, and overall trends of the growth of the system are also highlighted. However, no information is displayed about any relationships between the different classes, whether through a call graph or using inheritance. Provided the language being visualised did not support multiple inheritance, such as Java, then a simple modification to the visualisation would allow inheritance details to be shown. Limited call graph information could also be provided through the use of a call-stax like representation [Young99], without having to redesign the visualisation.

2.4.4.2.5. Visual Class Multiple Lens

The visual class multiple lens technique [Cain01] is designed to show how the usage of classes within large-scale programs has changed over a period of time. Class names are laid out using a radial tree layout, in order for the visual location of each class to remain approximately the same throughout the evolution of the program. Two metrics may be visualised with the technique. In the case study provided, colour was used to indicate the number of other classes used by the class in question, with red indicating a high value. The size of the class name was also used to represent the number of references of that class within the rest of the program.

Figure 2-4 demonstrates how the visualisation may be used to detect classes requiring attention, and highlights changes occurring over two years. Initially, the major class within the program is CWnd, which is referenced often by other classes. By the end of the two years, CMainFrame and CBlock are both large and therefore referenced frequently implying that changes to these classes will require significant recompilation. In addition, the fact that CMainFrame is represented in red highlights that it has probably become overcomplicated and is in need of refactoring.

Figure 2-4. Sequence showing change in usage of classes

2.4.4.2.6. Software Change

Eick et al. [Eick02] propose a number of new visualisations, based on data used within the code decay project (section 2.2.2.3), at both the code and project level. These new visualisations are based on various aspects of change data: the time of the change, the developer involved, the effort required in developer-hours and calendar time, the various files affected, the size of the change in LOC and affected modules, and finally the type of change that was made.

Five different metaphors were used, which were combined into a *perspective* to show various different aspects and relationships within the software. Often, the same data was displayed in more than one view, in order to highlight a different aspect. These views were also linked ensuring that any selection occurring in one view was shown in the others also.

A matrix view was used to show simple relationships, such as the number of changes made, indexed by developer and module. Simple cityscape views were also used to highlight the same information, although these were found to have no real benefit over the matrix view.

Bar charts and pie charts were used as the predominant view in the examples provided, showing the number of deltas against time, changes in lines of code, number of changes per year and many other attributes. These were found to be most effective when linked with other views, rather than when viewed in isolation. However, some problems regarding scale were highlighted for pie charts.

Data sheets (similar to a Table Lens) were used to show textual information about the changes. When zoomed out, the text is replaced with horizontal bars representing text, or the value of any numbers in the table. These were used in most of the views presented to show the low level details.

Finally, graphs (network views) were used to show any relationships between data items, although no real consideration was given to the scale or layout. Within the example, this was used to show files which had been changed as part of the same maintenance request.

The different perspectives provided allow the authors to answer several questions about the software, such as 'who wrote the code?', 'who changed which parts of the code?' and 'which files were changed together?'. By combining different views together, it is clear that other questions could also be answered.

Overall, this represents a valuable means of representing change data. By using simple generic views, the learning curve for a user of the system should be small. The power of the system comes from joining these views together in order to visualise more complex relationships. The danger is that by having a number of different views on screen simultaneously, it may be difficult to determine exactly what each view represents. This problem is likely to diminish with continued experience of the visualisation. The other problem with the system is that viewing 'physical' relationships – such as coupling – is difficult, and further views are required.

2.4.4.3. High level software visualisations

Various visualisation systems have also been developed to show a higher level view of software evolution than is available with the previous visualisations. This may be at the repository level, where the state of the configuration management repository is shown. Alternatively, changes may be shown at the module, rather than the file level, as is the case with SeeSys. A number of these interactive systems will now be reviewed. However, other static views for displaying changes exist. For example, Ball et al. [Ball97] use a clustered graph to show the files that were changed as part of the same maintenance request, and Eick et al. [Eick01] also use clustering to show modules that are often modified together, over several periods of time. Nodes are shown with a long tail, with the end point of the tail indicating where the node was in the previous time period.

2.4.4.3.1. SeeSys

SeeSys [Baker95] seeks to show changes at a higher level than a tool such as SeeSoft does, and deals with systems with millions of lines of code. It is based around a tree-map [section 2.4.2.3], splitting the system into subsystems, then directories, and finally individual files. It displays metrics that are both quantifiable and additive, using area, colour and shading. The large amount of information is placed in context by visualising one layer of the hierarchy at any one time, with the ability to drill down to the next layer when required.

The details are displayed by partially filling the nodes within the treemap. Both the ratio of the fill area to the rectangle area, and also the actual area of the fill allow easy comparisons to be made. The system also supports animation to see how the system changes over time. The rectangles are set at the start of the animation to be a constant size, and so the fill of the rectangles is used to determine the actual value of the metric. Support is included for mouse over operations to allow concentration on a particular subsystem/directory during the animation by showing further statistics separately. The advantage of setting the rectangles to be a permanent size is that no unnecessary movement is created during the playback. The disadvantage is that large amounts of space are wasted during the early stages of the project.

The major problem when dealing with a system of this size is that any subtle changes between small files or directories will not be seen. The reason for this is that the size of each rectangle is proportional to the size of the underlying system. Therefore, if there is one very large system and many smaller ones, most of the area on the display will be given to the large system, thus meaning that only a few pixels will be allocated to the display of each individual small system. Although a zoom function exists in the tool, there will be no indication that it is actually necessary to focus in on the system. This problem could perhaps be overcome by including an additional mechanism for highlighting nodes that have undergone change.

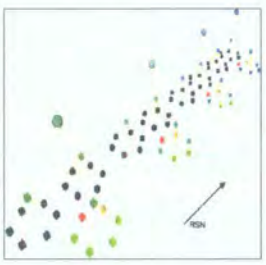
2.4.4.3.2. 3D SoftVis

3D SoftVis [Riva98, Gall99] also aims to provide a high level view of the evolution of a software system. The primary aim is to gain an overview of the evolution, in order to be able to make informed decisions about future developments. The visualisation concentrates on the module level, with no attempt to provide source code information. Differences are calculated using a 'Software Release History' method, which consists of examining the changes in structure (such as modules added or deleted) or in attributes (such as a module increasing in complexity). The release sequence number extracted from the version control system is used as the basis for the evolution over time.

Three aspects of a system are simultaneously visualised - the system structure, how it has evolved and metrics for the individual modules. In the case study used, the structure was particularly hierarchical, and so a cone tree representation [Robertson91] was used to display the overall structure for one system release. A 2D cone tree may be used when a historical view is required, with time then mapped onto the z axis, creating an alternative 3D view. In both cases, the required metrics are displayed using colour. This 3D view can then be mapped onto 2D, by removing the structural information and metrics information. This provides a very clear view of the overview of each module, with colour used to show the release number at which the module was last changed. This highlights the stability of modules, with modules undergoing many changes appearing in many colours. A number of images from this process are shown in Figure 2-5.

The 3D view is similar in many ways to SeeSys [Baker95], except that whereas SeeSys used animation in order to show the evolution of the system, SoftVis instead maps this onto a third dimension. This has the advantage of the entire evolution being visible at once, but causes problems with occlusion and navigation that are a potential difficulty with using any 3D system.

3-D visualisation of the structure of a system



The histories of entire systems can be viewed with multiple 3D trees.

An example of a multiple 2-D tree with percentage bars

Rotating sequential 2D trees creates a 2D table of the system history.

Figure 2-5. Example views from 3dSoftVis.

2.4.4.3.3. VRCS

VRCS [Koike97] is a 3D visualisation system developed using Vogue [Koike93], and is designed to be a front end for the tools RCS and Make. Figure 2-6 clearly shows the relationships that exist within a small repository. A graph drawn in 3D space is used to show the state of the repository - heavy lines show the evolution of an individual file, and light lines indicate how these files are associated particular

releases. Time is shown using the Z-axis. The visualisation is very clear on a small scale - it is easy to see that file A has gone through four revisions, with v.3 used for the second release of the system. However, previous studies with 3D graphs have shown that they have the same problems with scale as a similar 2D graph [Young99]. Therefore, the benefits of the visualisation when used with much larger projects are unclear.

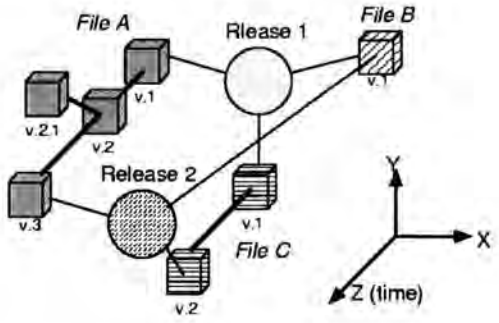


Figure 2-6. Two releases from RCS shown in VRCS.

2.4.4.3.4. Beagle

Beagle [Tu02] aims to provide a suitable environment for software maintainers studying the evolution of systems that have undergone architectural change. The main focus of the research concentrated on the analysis of structural changes. What is particularly interesting about this system is that it supports the renaming and moving of functions and files. This is important when analysing software that has undergone any reengineering, as it is likely that functions will be moved during the restructuring process. Failure to monitor this will instead record large numbers of new and deleted functions, and any consistency between the renamed functions will be lost. The technique works by extracting metrics about each function, and then using these together with the function names to match up functions from one version to the next. Dependency analysis is also used in order to verify the function matching process.

The visualisation itself is relatively simple. Files are shown within a 2D tree control, and can be expanded to show functions within those files. The icons are coloured to show whether the entities are new to the release (red) or are missing from the current version (blue). In addition, green icons show whether parent nodes contain information that has changed. This provides a useful information hiding technique by ensuring that information contained within an unexpanded node is not ignored. Comparisons can only be made across two versions at any one time.

2.4.5. Summary

This section has discussed a number of the benefits of software visualisation. These are based on reducing the mental effort of determining the precise relationship between multiple items and attributes within the data, and allowing trends and patterns within the software to be identified. A number of features that would be provided by an ideal visualisation were then listed. Finally, the section featured

a wide range of existing visualisations, demonstrating general data representation techniques and systems able to highlight software evolution.

The main drawback of the systems evaluated is that they do not provide detailed views of the evolution of the system over a number of releases. This makes it difficult to view how the individual relationships between files have changed, or to evaluate the short and long term effects of a modification made to the system over a period of time.

2.5. Animation

Animation may be defined as:

> *"The technique of filming successive drawings or positions of puppets or models to create an illusion of movement when the film is shown as a sequence."* [Allen90]

Roget first defined the concept of animation in 1824, with a paper entitled 'The persistence of vision with regard to moving objects'. The idea was popular, and lead to the introduction of the cinematograph in 1895 as a device for projecting moving images. Finally, computer animation was proposed as early as 1964, with Knowlton starting to develop techniques for animation at Bell Labs [Vince00].

The underlying concept behind animation is the persistence of vision. The human eye takes about 25ms to convert light to electrical signals to be processed by the brain. Therefore, if static images are presented to the eye faster than about 25ms then smooth movement is perceived, although the exact value is dependent on the brightness of the images used. Images presented at more than 200ms apart cancel any apparent motion [Bartram97]. Other factors are also involved. For example, the 'Phi phenomenon' involves a darkened room, with two bulbs placed within a short distance from an observer. By switching the lights on and off alternately, the observer sees a single light that moves from one bulb to the other. The reason for this phenomenon is not yet known. Additionally, the realism of the scene is also important. Within a realistic scenario such as a detailed 3D environment, faults such as an incorrect walking motion for avatars are easily identified and cause disruption. Within a less detailed environment, the visual system is much more forgiving.

Within computer animation, there are a number of techniques used to reduce the amount of information required by the animation system. *Keyframing* reduces the number of frames that a user must draw to achieve smooth animation. Instead, the important frames are provided to the system, with objects at a set colour, shape, texture or location, together with the frame number at which this must occur. The animation system then goes through a process of *inbetweening*, where the frames between two keyframes are generated automatically to provide smooth movement. This process will rely on a series of *interpolation* functions, where function curves are used to provide smooth motion. For example, rather than an object moving at constant speed from A to B, mapping the speed onto a sine wave will provide a more realistic slow-in, slow-out motion. Similarly, motion *paths* are used so that instead of moving from A to B in linear fashion, the object will follow a predetermined route such as a curve.

2.5.1. Algorithm animation

Within the field of software visualisation, any use of animation will predominantly refer to 'algorithm animation'. Although algorithm animation has traditionally been seen as part of software visualisation [Roman93, Price93], recent research considers algorithm animation as a very different research area [Maletic02]. Stasko [Stasko90] defines it as:

> *"... the process of abstracting the data, operations and semantics of computer programs and then creating animated graphical views of these abstractions".*

Algorithm animation will usually be done on a very small scale. The purpose is usually educational with the motivation of teaching novice programmers how various algorithms work, by showing the movement of data within the algorithm itself. In most cases, algorithms must be modified by hand in order to provide the animation system with details of the location and values of the various data items under consideration. For this reason, there is no expectation for algorithm animation to consider many of the problems of software visualisation such as automation and scale. However, it still provides some useful and practical examples of using visualisation techniques together with animation. Some systems will now be examined from the perspective of creating and displaying animations.

2.5.1.1. Zeus

Zeus [Brown92] is an algorithm animation system designed for both users and programmers, and is an extension of an earlier system named Balsa [Brown85]. Programmers can animate algorithms by including 'interesting events' – a procedure call - at particular points in the algorithm. These interesting events call the display routines within the system using the data provided. The programmer also needs to provide an event file, which summarises the important values that should be monitored in the algorithm.

The actual display can be implemented in one of two ways. The first is a simple transcript that is generated automatically from the event file. Secondly, a simple editor can be used to create 'building blocks' that may be used later when coding the animation. These blocks may then be further customised when the animation executes, such as by setting the location and colour of an object within the block. Animation is achieved by ensuring that the system generates views that correspond to the current state, rather than the individual blocks moving or changing colour directly to reflect the current state of the algorithm. During the playback of the animation, the user may stop, start, change the speed and single step through the algorithm. Several views may be displayed simultaneously, and an upgraded system [Brown93] also supports sound and 3D graphics.

2.5.1.2. Polka

Stasko et al. took an alternative approach with TANGO [Stasko90] and then POLKA [Stasko92]. The primary difference between POLKA and Zeus was the idea that program objects should be associated with graphical objects. In addition, this allowed support for the smooth movement of objects. Whilst

smooth animation could be introduced within Zeus, it required the animator to do the inbetweening process manually.

A program to be animated firstly requires an 'Animator' object dealing with event registration, and secondly the provision of a number of views. Events are inserted into the algorithm in order to update the animation as required in a similar way to Zeus. Graphical objects can be inserted into views at a given frame number, and grouped with other objects. These objects contain a shape, a location and an action, such as 'move' or 'colour', all of which can be changed over time. Interpolation is linear, with the animator providing the number of frames that the change must take place in. The exception to this is shapes, which change discretely. The animation is programmed in advance, rather than immediately executed, meaning that parallel animation of different objects is much easier to achieve. This requirement means that for the same animation, more work is required within Polka than it is in Zeus.

2.5.1.3. Eliot

Eliot [Lahtinen98] attempts to automate algorithm animations using the animation power of Polka. A tool developed by LaFolette et al. [LaFolette00] attempts to do the same, but with their own animation library. In both cases, the tools aim to analyse the code semantically in order to generate the algorithms. For example, declaring an integer of type INT rather than int indicates that this object should be animated. If this integer ever appears in a tree or list, or as part of an array, the system will display the relevant data type, and animate it as the content of the data type changes. It is clear that such an approach is much less flexible than creating the animations manually, and also that the tool itself is significantly more complicated. Much work is required in this area in order to increase the range of data types supported for automatic animation.

2.5.1.4. Empirical Evidence

The underlying assumption behind algorithm animation systems is that a better understanding of the algorithm will be achieved by viewing the animation. This is a view shared by both creators and users of the animations [Lawrence94]. However, there have been a number of studies that indicate that the assumption is not necessarily a valid one.

Lawrence et al. [Lawrence94] studied the effect of using algorithm animations instead of static slides for teaching a spanning tree algorithm during a lecture. They also examined whether there was a difference between students who were given the data files necessary to feed into the system, and those who had to create the required files within a laboratory session. The students were tested with a short exam. The results showed that within the lecture setting, there was no significant difference between the use of slides or animations. However, within the laboratory setting, there was evidence that interacting with the algorithm animation system was beneficial to learning, with increased benefits if students were allowed to provide their own data.

Byrne et al. [Byrne96] examined whether algorithm animation would help students predict the behaviour of a given algorithm. The prediction aspect was considered to be important, as it was felt that

this would indicate better understanding of the algorithm. Using animation would allow these predictions to be verified as the algorithm progressed. The benefit of prediction was found as a result of the first experiment, where students were encouraged to predict behaviour regardless of whether animation was used or not. It also showed there was a small benefit to using animation. The algorithm used was a simple depth-first search, and it was tested on non-computer science students. The experiment was then replicated using a binomial heap algorithm with computer science students. In this case, there was no benefit found for either animation or prediction.

Kehoe et al. [Kehoe99] also researched the role of algorithm animation within teaching. Unlike Lawrence and Byrne, where the students were tested with an exam, Kehoe et al used an open book approach, where the materials provided could be examined and reviewed at any point. No time limit was set. Twelve computer science graduates participated in the study, all with algorithm experience. The subjects were split into two groups, with half provided with animations, and the other half with static screen shots from the animation tool used. Both groups also had access to text and pseudocode describing the algorithm, which was the same binomial heap that was used within Byrne's experiment. The results were significant, with a benefit for the animation group.

All these studies show that animation will not automatically improve learning and understanding. In particular, by preventing interaction with the animation, students were thought to just be observing the proce ss, rather than understanding it. Similarly, the best result for algorithm animation came when students were allowed to use animations together with other materials in an unpressured setting.

2.5.2. Other uses of animation within visualisation

Although the use of animation within software visualisation has concentrated on algorithms, there has been some work to extend this into other areas, for example to show data flow within UML diagrams [Burd02]. However, animation also has wider application areas than just showing the progression of time. Cone trees [Robertson91], for example, use animation as a data hiding mechanism, by rotating the cones as necessary in order to display the current path to the root node. Many tools will use animation in order to make smooth transitions when scrolling or zooming [Jog95, Benderson98], or during layout changes [Yee01].

Animation can also be used to show other variables. Ware and Limoges [Ware94] examined the use of oscillating, rather than static, items in a visualisation. Three relevant attributes were considered which were the amplitude, the frequency and the phase of the oscillation. However, phase was only relevant if the frequencies of the objects were the same. These could be applied in both the x and y dimensions. Experiments showed that phase was the most useful property when displaying correlation information, and possibly preferable to colour or size. Amplitude and frequency were not considered useful. The main problem with using phase is that determining the actual position of an oscillating point is more difficult.

Bartram [Bartram97] suggests other properties that could also be used to group related objects together. These include the use of relative velocities and trajectories, transitions, and attraction and repulsion.

Also, movement may be interpreted by the user as being emotive, such as urgent, locomotive or expressive, which could also be used to classify objects. Some experiments examining the role of paths [Bartram01] showed that the shape of the motion was an excellent feature for both grouping and evaluating. This was true even when a large distance separated the points. Phase was also found to be useful, but to a lesser extent.

Finally, animation may be used in order to draw the attention of the user to a particular part of the visualisation. Bartram et al. [Bartram01b] studied the effectiveness of motion or colour change of an icon in order to attract the attention of a user engaged in a separate task. Their conclusion was that motion was the more effective means of attracting attention. A second study involved determining how distracting and irritating the different forms of motion were. They discovered that large amounts of movement involving significant tracking were most irritating, but small amounts of anchored movement, where the object moves linearly, and jumps back to the original position, was least distracting.

2.5.3. Cognitive issues regarding animation and visualisation

Although there have been a number of studies examining the effectiveness of animation with teaching algorithms, there has been less research in other areas of visualisation. In particular, Scaife and Rogers [Scaife96] state that no model has been developed for the role of animation in either a learning or a problem solving context, and it is unclear how animation assists in the process. They also question the common assumption that adding animation to an equivalent static diagram will provide more accurate information, and reduce the processing demands required. Instead, they write that:

> *"As with diagrams used in specialised domains, e.g. physics, ... a person has to learn to 'read' and comprehend the significance of the content of the animations in relation to other information that is being presented. This requires making multiple connections between what the animations are intending to convey and the abstract concepts that are being learned about."*

The important point here is that users must learn to read the animation, and so can not be guaranteed any instant benefit from looking at an animated diagram. This would seem to support the results from the algorithm animation experiments, where the greatest benefits were achieved when the users were interacting with the systems, and providing alternative data to view the overall effects.

A second issue regards the concept of 'change blindness' with animation, which is of particular significance to time-based visualisations. A very common assumption is that given two images overlaid on each other, such as frames within a longer animation, any changes between those two images will be spotted easily. However, a series of recent studies has shown that this is not the case [Simons00, Rensink02]. These studies show that even large changes will go undetected in certain circumstances, and this has some impact on the role of using animation for visualisation. The usual example is where a blank screen is shown for a very brief period of time between the two images, or the images are interchanged during the blinking of the eyes. However, change blindness is also apparent when splashes are shown on the screen for short periods of time, when the locations of these splashes do not

correspond to the locations of any changes. Finally, objects fading very slowly in and out of images, or slowly changing colour, may also go unrecognised. These changes must take place over a series of several seconds if they are to go unnoticed. There are a number of theories as to why this occurs, although no definite answer is known. Importantly, once the observer is made aware of the changes, they are spotted easily. Therefore, indicating the location of the change beforehand in some obvious manner can negate the effect. Alternatively, flipping between two views rapidly with no blank in between, assuming the locations of the objects remains the same, is also successful [Fekete02].

2.6. Summary

This chapter has examined a number of different areas of research. Software evolution states that successful software will continually undergo modification. Configuration management systems aim to store the various software revisions that are produced during the modification process, in order to make them easily accessible at a later point in time. Visualisation is a recent area of research that seeks to use images to comprehend large amounts of complex data. Finally, animation is a means of displaying processes in an intuitive and successful manner.

It can be seen from this that there are a number of overlapping areas. Software evolution states that as programs evolve, they will become larger and more complex. Software visualisation aims to help developers and managers comprehend large and complex software systems, by viewing relationships between various aspects of the software. As can be seen from the various taxonomies, there are a number of different approaches for doing this, all with different motivations and results. Therefore, it seems logical that some form of visualisation technique should be beneficial when working with evolving software.

Open source software presents an extreme view of this, with a general emphasis on small, frequent releases. The nature of development means that changes will occur regularly within the software. It is also often the case that changes will be submitted without prior change requests or assignments. This makes keeping track of changes, and thus the current state of the project, difficult for both developers and managers.

Current configuration management software provides very simple tools for dealing with this problem. Diff, and the various derivatives of it, are designed for use by the configuration management system to store deltas in an efficient format. With very few other tools available however, maintainers must also use this tool. Although more user-friendly representations of the output exist, for example with CSDiff [CSDiff03], the main problem is that the tool was not designed for use in this manner. Therefore, changes are shown on a line-by-line basis, with no consideration for syntax or modifications spanning a number of files. Better options are available, such as comparisons taking a syntactic or metric based approach.

Therefore, given the relevance of visualisation to evolution, it is a small extension to develop visualisation tools with a greater emphasis on open source software. In particular, by basing these tools

on more user-friendly comparison tools, more useful and relevant results should be generated. This will hopefully ease the maintenance task for software developers.

Research has also indicated that there is a vast amount of information contained within configuration management repositories [Ball97]. Even without examining the actual source code, log files, comments, and even timestamps can be used to generate a picture of the current state of the software project. Visualisation should also be useful in this respect, by allowing further examination and exploration of this data. Therefore, project managers should also benefit from any tools developed.

Finally, animation is a natural and intuitive, means of showing the passage of time. Current uses within visualisation have been restricted predominantly to showing data flow in a very limited fashion through the use of algorithm animation. However, this could be extended to show how the program itself has changed over time. By using animation in this way, the visualisation becomes more than just a teaching tool, but instead a viable means of showing evolution within software of a far larger scale. Obviously, as the research on change blindness shows, this must be done with care in order to remain useful.

Although these overlaps are obvious, there are very few examples where tools have been developed combining these aspects. Therefore, based on the individual areas of research, the remainder of this thesis will examine the effects of combining these areas. In particular, although a case was made for combining visualisation and evolution, there are a number of new problems that this creates. This is in addition to the various difficulties that visualisation must solve, such as scale or navigation. The next chapter will investigate some of these problems, and various solutions to them. Some of these solutions will then be examined in later chapters, by developing visualisations with the aim of highlighting changes within evolving software, at various levels of detail.

ANIMATING THE EVOLUTION OF SOFTWARE

3. Issues involved with Visualising Evolving Software

3.1. Introduction

Software visualisation has advanced significantly from its original roots in flow charts. New visualisations will often use novel techniques in order to display the information contained within a set of data in an accessible manner. Various guidelines exist as to how to maximise the potential from using different representations and layouts. Recent work has also examined the benefits and drawbacks of using 3D within visualisation.

Many of the existing software visualisations are based on visualising properties of the current state of the software, such as the control flow or the complexity. However, there are an increasing number of visualisations that incorporate change history information, such as changes to the size of the source code, in order to visualise some aspect of the evolution of the software over a period of time. This chapter aims to investigate a number of issues involved in this process.

3.2. Definitions

For the purpose of this thesis, it is necessary to define a number of terms with respect to the data being visualised, and the behaviour of the visualisation. These are defined as follows:

Static Data Data that will always be constant at any point in time, such as the complete source code implementation of a specific bubble sort algorithm.

Dynamic Data Data that is constantly undergoing modification, such as the array involved in the bubble sort algorithm as the algorithm executes.

Evolutionary Data A sequence of static data, where each entry represents the data at a specific point in time. Furthermore, this sequence will continue to expand over time. For example, consider a complete series of execution times of a bubble sort algorithm for a random data set of fixed size. Each time the algorithm is executed, the new execution time will be appended to the sequence.

The static data contained within evolutionary data taken from an earlier point of time will also, by definition, be contained within the evolutionary data taken at a later point.

Static, dynamic and evolutionary visualisations may now be defined as being targeted towards static, dynamic and evolutionary data as follows:

Static Visualisation A visualisation designed for static data. Once the data set has been included by the visualisation, no further updates are possible.

Dynamic Visualisation	A visualisation designed for dynamic data. The visualisation must therefore handle a changing data set during the execution of the visualisation.
Evolutionary Visualisation	A visualisation designed for evolutionary data. Once the evolutionary data has been included by the visualisation, no further updates are possible without restarting the visualisation. However, the data should be recognised as evolutionary, in that later items within the sequence will be related in some way to the previous items.

There is a close relationship between evolutionary, and static and dynamic data. For example, evolutionary data may be generated easily from dynamic data, by storing all of the values of that data as it is modified, over a set time period. By freezing this data at set points in time, the data may then be treated as a static data set.

Based on these definitions, there are many examples of static visualisations These include SeeSoft [Eick92] and SHriMP [Storey01] to name but two. There are also many examples of dynamic visualisations. A primitive example is an algorithm animation system, where additional code is embedded within the algorithm allowing the visualisation to reflect the current state of the algorithm at that point. More complex systems also exist, such as DJVis [Smith02], where the visualisation is closely integrated with a debugger, allowing the user to step through the code observing changes as they occur. Finally, Software World [Knight99] is an example of an evolutionary visualisation, where changes to the software are shown using alternative colours and textures. SeeSys [MBaker95] is another, where each file that has existed at some point during the project is mapped to a rectangle, and the state of the files for a given item within the data set may be shown.

In addition, the following visualisation terms will be used throughout this thesis, and will be defined as follows:

Data Set	The complete data given to the visualisation (e.g. releases 1 – 15 if evolutionary data).
Data Item	Within an evolutionary data set, the data relating to a specific time. (e.g. release 2). If the data set is not evolutionary, then the data set will contain a single data item.
Data Entity	Part of the data within a data item (e.g. file "hello.cpp" from release 2).
Element	A graphical object within the visualisation that reflects the underlying data entity.
Representation	The graphical representations used to represent specific data attributes and values.

Mapping	The process of going from a single data entity to an element by use of the representation.
Layout	The algorithm for arranging elements within a view.
View	The overall image, of which the display will be a subset, made up of all required elements, and set out using a layout.
Display	The image displayed on the output medium.
Interaction	The means of interacting, through mouse or keyboard, with a view.
Metaphor	A set of representation, mapping, layout and interaction that together mimic real-world behaviour.
Visualisation	A collection of associated views and interactions.

Finally, it is necessary to define two different forms that the view may take, in order to consider the impact of dynamic or evolutionary visualisations.

Permanent View	A view where, after the initial view is generated, elements may not be added, removed, modified or relocated within that view without direct intervention by the user.
Transient View	A view where, after the initial view is generated, elements may be added, removed, modified or relocated within the view without direct intervention by the user.

A dynamic visualisation will often require a transient view. This is because the data will change during the execution of the visualisation. Therefore, in order for the view to continually reflect the state of the data it will be necessary for the elements representing the data to be modified. Although it is possible to allow the user to confirm every request for the visualisation to be changed, thus creating a permanent view, in most cases this behaviour would be highly disruptive.

An evolutionary visualisation may conceivably use either view type. If the individual items within the data set are small, then the visualisation may show the entire data set simultaneously using a permanent view. For example, 3dSoftVis [Riva98] and the Evolution Matrix [Lanza01] use this approach. If, instead, the visualisation uses animation to show a series of items within the data set, with only one item visible at any point, then a transient view is required as demonstrated by SeeSys [MBaker95] and Time Tubes [Chi98].

Static visualisations may also use either view type. In most cases, there is no benefit from using a transient view. However, if the data includes any time-based sequences, then a transient view could be used to highlight this within the display. For example, Burd et al. [Burd02] use animation to show the order of a sequence within a UML diagram.

3.3. Evolutionary Visualisations

Transient views are often necessary for evolutionary and dynamic visualisations in order to represent the data most effectively and unobtrusively. However, the use of these views presents new challenges and problems for software visualisation. Shneidermann's seven-point plan [Shneidermann96] details a series of guidelines for a visualisation, concentrating on zoom, details-on-demand, and other features. However, this is very much based on the use of a permanent view. For example, consider an overview display showing all of the elements within a visualisation. A zoom-in operation will allow more details of some of these elements to be seen, but reduce the number of elements shown. This poses no problems for a permanent view, as no changes can be made to the area outside the new display that will go undetected by the user. However, this is not the case when a transient view is considered. Here, an element within the visualisation could be introduced and then removed from an area outside of the new display. As the user saw neither the appearance nor disappearance of the element, they would be unaware that the element ever existed. This obviously has some impact on the accuracy of the visualisation.

Similar issues are raised by the use of evolutionary visualisations using transient views, or dynamic visualisations using either view. For example, the bookmarking functionality that is suggested by Storey [Storey97] does not consider that the bookmarked element within the visualisation may not exist at the current point in time. Design guidelines and evaluation frameworks do not currently consider these issues although it is possible, to some extent, to extrapolate new guidelines for these situations. For example, bookmarking could involve both the elements under consideration, and the state at that point, allowing the user to jump to either the element in the current view or in the original view. However, as new work continues within this area, there will be an increased need for dedicated research into transient views, and evolutionary and dynamic visualisations.

The following sections will examine three further issues related to evolutionary visualisations. Although these issues may also be relevant to dynamic visualisations, they will be considered from an evolutionary visualisation perspective. The importance of maintaining consistency, or familiarity, during the visualisation will be addressed together with a series of strategies for how this may be achieved. Similarly, the relevance and problems of using a metaphor within the visualisation will also be considered. Finally, animation, which is a common method for displayed time-based data, will be considered. However, with the exception of algorithm animation, there are very few examples of the use of animation within the field of software visualisation. Therefore, some of the benefits and drawbacks of using animation will also be considered, together with some possible reasons as to why animation is so infrequently used.

3.4. Familiarity

It is important for any visualisation to be able to produce two identical views when given two identical data sets. The reason for this is that although a visualisation can present an overview of the data in a very short length of time, further study will take a significant investment of resources. This study will

involve identifying relevant elements within the visualisation, through naming, positioning and colour, in order to be able to draw conclusions as to the behaviour of the underlying data. Therefore, in order to preserve this investment, it is important that the visualisation may appear to be identical to the last time that it was viewed.

Obviously, there are occasions where this behaviour is not always required. By presenting the data from a different perspective, for example, by changing the layout or representation, patterns may be spotted in the data that were not obvious before. However, the point is that the visualisation must at least support the provision of a familiar view for the user. Elements within the visualisation should therefore be coloured appropriately, and located in the positions expected from the user's previous experience.

Within a static visualisation, achieving familiarity with the same data set is trivial, as a view may be saved at the end of a session, and recalled for the next. Even if this is not the case, as long as systematic layout algorithms are used, familiarity should always be achieved.

However, the situation becomes much more interesting when the idea of evolutionary data is introduced. There are few situations where it will be necessary to visualise a completely static data set. Rather, it is more likely that the data will undergo continual modification. This could be the result of changes in the structure of the data, the number of data entities, or the actual values held within the data. Therefore, it is necessary to take the idea of familiarity for static data sets, and expand this to deal with evolutionary data sets.

Although this concept is applicable to visualisation in general, the issue is particularly evident within software visualisation. Young [Young99] comments that:

> *"When changes are made to the underlying software system, it is imperative that the new visualisation changes as little as possible to reflect this. This resilience of change is necessary in order to maintain the user's mental model of both the software and the visualisation. If the structure and layout of the visualisation were to change drastically then the user would need to relearn and explore the environment each time. This is clearly counterproductive."*

Change is inherent in software. As mentioned in section 2.2.2.1, a program that is used must be continually adapted else it becomes progressively less useful. [Lehman85]. Therefore, when visualising software, it is almost certain that that software will have been modified in some way since it was last viewed. It is also the case that the program will become more complex over time. [Lehman85]. Therefore, it is not usually possible to add any new changes onto the 'end' of the visualisation, as is possible with some data. Share prices, for example, are constantly changing. However, the latest price can always be placed at the end of the graph, whilst still maintaining a clear and familiar picture. With software, this is less likely. A change can not be guaranteed to be at the fringe of a program, but rather could be a modification of a core class or data structure. Even if the change was at the fringe however, it could create a new call to a central module, which may result in a radically different layout for a call-graph based visualisation.

The large number of relationships contained within software also means that visualisations tend to be significantly more complex than a simple line graph. A change to the data could affect not just the layout used within the visualisation, but also the representation. For example, if colours were allocated alphabetically to class names for quick reference, then inserting a new class at the top would result in every other class having a different colour, with the result that familiarity would be lost. It has even been suggested that a modification to the software could result in a different metaphor being required. [Knight00b].

Given that it is important for familiarity to be maintained with evolutionary data - or for the visualisation to have good 'resilience to change', it is necessary to investigate the benefits and drawbacks of some existing methods in order to determine to what extent this property can be completely satisfied.

3.4.1. Strategies for handling familiarity

One of the most critical parts of a software visualisation is the layout algorithm used. The layout should allow elements to be identified easily, and in the correct context to other elements. It is also the most difficult part of a visualisation in terms of the resilience to change property. Ideally, each layout for additional data should provide an optimal placement of elements, that is as similar as possible to the previous layout. Small changes to the data should have a small impact on the layout of the elements. Obviously, in many cases, there may be no such thing as an 'optimal' placement. Also, maintaining a high degree of similarity is not a trivial task.

In an attempt to achieve this, five strategies will be proposed as a means of providing some form of resilience of change to a layout.

3.4.1.1. Omniscience

Omniscience is the most basic strategy for achieving the resilience to change property. It supports evolutionary data, but only by ignoring the requirements for further expansion of the data.

If a complete data set is provided at the start of the visualisation, a suitable layout may be generated by merging the information contained within each data item. For example, if a graph based visualisation was developed then a graph could be created using all of the nodes and all of the edges that existed at any point within the evolved data set. This graph could then be run through a suitable layout algorithm to produce a master layout.

Visualising the evolution of the data is now a simple task. Any nodes or edges that exist during the current 'time' are displayed, whilst any others are hidden. As the visualisation continues to play through time, 'new' data will be displayed by revealing the relevant nodes and edges. In the same way, 'deleted' data is handled by hiding existing nodes and edges. Therefore, there is never any danger that the visualisation will run out of space, or that adding a node in the centre of the graph will cause significant layout changes. No nodes will ever have to move to accommodate new or deleted

information, and so the layout will remain as static as possible. A small change to the data, such as removing a node, will result in a small change in the visualisation – the node will be hidden.

This is obviously a very desirable situation. However, there are two significant problems with this approach.

Firstly, there is no concept of '*future proofing*'. Although there is a perceived resilience to change in the visualisation as minimal change occurs when moving through time, this is only the case for historical data. If the data set is then expanded at a later date, and the visualisation used again, a different master graph will be generated. Depending on the algorithm used, this could have huge effects. For example, consider a visualisation where a simple layout algorithm is used, and nodes are laid out alphabetically. If several new classes are introduced, the new master layout may be significantly different, particularly if all the new classes start with 'A'. Therefore, although the new master layout will display the new, expanded, data set appropriately, there may be no correspondence to the old layout, and any information regarding this old layout will have to be relearned for the new one.

The second problem is that the view will normally be very unbalanced during the early parts of the visualisation. Programs will almost always grow over time, and so the number of nodes existing at the end of the time within the data will significantly exceed the number at the start. Therefore, at the start, the graph will appear very sparse, possibly with huge spaces between each node, and will be difficult to view. At the end, the graph will appear denser, but there will still be unnecessary gaps, as deleted nodes will still be taking up space. (There may, however, be some cases where these gaps are desirable, depending on how interested the user is in examining the reasons for the deleted data).

The concept of future proofing is vital to achieving full resilience to change. The following methods will all therefore contain some support for future proofing.

3.4.1.2. Big Box

A 'Big Box' strategy is similar in some ways to the omniscience strategy, but with better support for future proofing. The idea is based on the concept of a 'future-proof' personal computer. Traditionally, a desktop PC has always been seen as more future-proof than a notebook PC. One of the reasons for this is that a desktop will have a larger case, more available expansion ports, and so on. However, the desktop is also significantly larger than the notebook, in order to reserve space for adding new peripherals. Initially, most of the inside of the PC will be empty space, which may fill up over time as new devices are added.

This space is also pre-assigned. For example, adding a fifth hard disk if only four drive bays are provided in the case is impossible, regardless of how many generic expansion slots are still free. Therefore, in order to add the new hard disk, it will be necessary to throw away the old case (with a lot of free space remaining), and obtain a new, larger case.

The analogy applies very well to visualisation. The idea is that when generating the initial layout from the given data, far more space than is necessary is allocated to each element, thus leaving plenty of space for future expansion. As new data items are introduced to the visualisation, the new elements can be placed in the previously allocated space, and so further layout changes are unnecessary, as demonstrated in Figure 3-1. This is the approach used by Software World [Knight99].

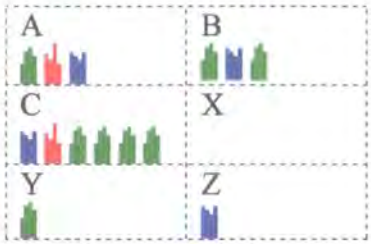


Figure 3-1. Initial big-box layout, preserving extra space.

However, once again, there are significant problems to this approach. The first is that the strategy only works effectively with an even distribution of data. As with the PC analogy, unevenly distributed data means that one area of the visualisation runs out of space due to excess data, whilst others remain empty. A possible solution to this is to carefully weight the free space allocated. For example, if the data was laid out alphabetically according to class name, then it is reasonable to assume that space would be allocated alphabetically. In addition, more space would be allocated for classes beginning A, B, C than for those beginning X, Y, Z, as it is less likely that there will be a large number of classes starting with Z for example. However, such a distribution can then provide some unusual clustering effects, as demonstrated in Figure 3-2. Here, 'C' is allocated much more space, at the expense of X, Y and Z.

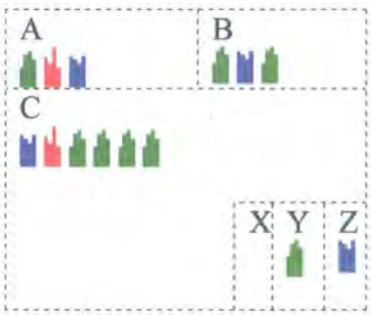


Figure 3-2. Clustered big-box layout, allowing additional space for 'C'.

However, due to Gestalt effects, it is now very difficult not to perceive X, Y and Z as a group, as they are so obviously distant from the other classes. In almost all situations, this would be a very unsatisfactory result.

A possible solution for overcoming this problem is to always layout the data chronologically. In this way, all the free space will be clustered together, and so will maximise the potential use of this space before the box is filled. This approach has the drawback that deleted data will leave spaces that will never be refilled. Also, it may be that a chronological ordering is less useful than an alphabetical ordering, particularly for navigation and browsing, although a simple 'find' function will go some way

to alleviating this problem. In many cases though, classes will be prefixed or suffixed according to module or operation, and so identifying classes in the same module will not be possible purely from layout alone anyway.

The other issue of a big-box strategy is that unless the box is of an infinite size, which may be possible in some situations, the box will eventually be filled. This means that a new box will have to be created, and the information transferred across. When this happens, a balance has to be struck between maintaining a similar layout to the old box to maintain familiarity, or rearranging the contents in order to remove deleted information, changing the distribution, and so on. The 'best' approach will vary depending on the task and data.

3.4.1.3. Abstraction

An abstraction strategy relies on increasing the data density displayed in the visualisation. By abstracting as much new data as possible within existing data, significantly more information can be displayed without increasing space requirements. This is important as this means that the same layout can be used as the visualisation evolves, and so layout familiarity may be maintained. As with a big-box strategy, the 'future' data does not have to be known in advance. For example, the sequence of images below in Figure 3-3 shows a section of code moving from one class to another. Colour is used to indicate the size of the class – small, medium or large. In both the start and end cases, the size of the class within the representation has remained constant, and there is no need for any layout change.

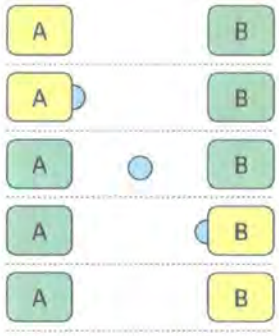


Figure 3-3. Example of code moving from class A to class B. Both A and B remain a constant size.

Again, there are some problems with this strategy. Abstraction can lead to important data being hidden or treated as insignificant, and the visualisation could easily become misleading. In particular, small changes that occur may not be apparent depending on the abstraction mechanism. To some extent, this can be avoided by temporarily highlighting any changes that occur within the data, even if the final view is identical to the previous one.

Also, abstraction is only a viable strategy at a certain level of detail. For example, if a class is used as an element within the view, then changes within the class can be incorporated easily. However, adding a new class reintroduces the problems of the previous strategies, unless the two classes are integrated

within the visualisation and shown as one. This may be appropriate in some cases, such as with abstract and implementation classes, but overuse will soon make the visualisation meaningless.

An extension to abstraction techniques is provided by semantic zooming [Benderson98,00], or virtual spaces [Knight00a]. The idea of these techniques is that it is possible to zoom in on a node, or enter a 3D structure, and then access further elements contained within it. Importantly, the space required for these elements does not need to correspond to the size of the original node. This therefore provides a layout with infinite space, where elements may be added without affecting the original view. However, this new space will also require a future proof layout. The main drawback of this system is that it assumes that each virtual space is self-contained, and so it is then difficult to show relationships between two elements contained within different virtual spaces. However, if the number of such relationships is low, the situation can be resolved through the use of additional windows, showing the original view and the virtual space simultaneously.

Provided the above pitfalls are carefully considered, then abstraction and virtual spaces do represent a useful way of postponing, although not solving, some of the problems related to a big box strategy.

3.4.1.4. Animation

Animation is a very useful tool for achieving some form of future proofing. This is because it may be used when changing from one layout to another, which is almost inevitable at some point during a future proof visualisation. The main reason for this is that it has been shown to aid in maintaining context, which reduces the relearning required to understand the new layout.

However, the use of animation to maintain context can not apply in every situation, as is sometimes assumed. Context will only be maintained if it is possible to track each and every node and edge to its new location. With large numbers of changes, this will not be possible. For example, during a morph of a human's face to a lion's face, it is not usually possible to see exactly how the process works unless viewing it slowly frame by frame. The result is seen as a smooth morph from one to the other, but unless the user focuses carefully on a particular area, they are unlikely to gain any additional information. This is also true when morphing graphs. With a large number of moving nodes, maintaining context is not possible unless viewed many times.

A second problem is that the human eye will be drawn to movement, particularly if that movement is sudden or extensive. This may be a serious problem, depending on the layout used. For example, consider a nodes-and-arcs visualisation. If the addition of one node causes a lot of other nodes to be moved to make space, it will be the other nodes moving that capture the attention of the user, rather than the fact that a new node is appearing. By moving attention away from the new node, and instead causing the user to focus on redundant movement, the visualisation becomes less effective.

There are some possible solutions that will go at least some way to resolving these issues. With the first problem, it is important to minimise the number of nodes that will move at any stage. In particular, if nodes can be grouped in some way, and then moved simultaneously to a new destination, the amount of

perceived movement is lessened. This is the approach taken by Gnutellavision, where a radial graph is used [Yee01]. However, there are very few other graph layouts that support operations such as this. If a large amount of movement is required, for example, due to a very significant change in the data set, then animation can still be used. However, rather than maintaining context, it should be recognised that this will probably break context instead. This is not ideal, but still important. If a user of the visualisation is accustomed to always seeing a particular class at the top left of the screen over a large period of time, unless it is made obvious, they will continue to think that this is the case. Animation can be used to make it clear to the user that the layout has been changed significantly, and they will have to relearn at least some of the visualisation. Obviously, playing the animation repeatedly might mean that some context will be regained, but the first impression should be that things are significantly different.

A solution to the second problem, that of drawing unwanted attention, is that the animation should become a two-stage process. Rather than moving the nodes and introducing the new node simultaneously, the two operations should be done separately. Firstly, the nodes should be moved. Then, once it is clear that the uninformative movement phase has ended, the new node should come into view. Fading the node in may give the user more warning about the change, rather than the node suddenly appearing. If nodes must also be deleted, then a three-stage process should be used.

3.4.1.5. Prediction

A prediction strategy is the most difficult solution to apply. The theory is that by observing how the data has changed previously, it will be possible to predict to some extent how it will change in the future. This would be very useful for a future proof visualisation. With the previous strategies, as soon as new data is encountered, it must be displayed by the visualisation, possibly causing significant layout changes. A prediction strategy would anticipate that space will be required, or could be reclaimed, at various points in the future, and so make more gradual changes to the layout in order that when the new data is encountered, space has already been allocated for it.

The most significant problem with this strategy is the extent to which it is possible to predict the future requirements. This will obviously vary hugely with the domain. Within software, Lehman's laws state that a software project will develop its own dynamics, which could be used to monitor and predict the evolution of the project in some way. Most research in this area concentrates on the cost required to implement a change within the current system, or the ease with which the system can be evolved. [Ramil00] However, recent work indicates that it is also possible to predict with reasonable accuracy the increase in LOC and number of modules that will occur within the next year [Caprio01]. Most software evolution studies have also been based on traditional developmental methods. As there are indications that open source development will not always follow the same patterns, a number of different prediction strategies may have to be included within the visualisation.

A further difficulty is raised with the fact that software visualisation will often be used purely because the system as it stands is too complex to be understood normally. Therefore, it is not possible to assume that the visualisation is dealing with 'normal' data. As Lehman discovered [Lehman00], a very large

number of changes within a release will make the management of the project during that release much more difficult, significantly increasing the number of future fixes required. This would also indicate that any predictions made may not be appropriate when the system reaches this critical stage, and so the visualisation becomes significantly less useful.

However, there are some benefits to the approach. In particular, it is almost advantageous to the user if the predicted requirements fail to match up to the data. If, for example, it is obvious that a space has appeared in the centre of the system, but that no new node has yet appeared within that space, it may be worth investigating what caused that space to appear, or why no new node has been introduced. This may be as a result of management changes restricting access to that area of the system, or because development has been concentrated on a different module. Similarly, if there is a great deal of sudden movement rather than a gradual change, this would indicate that significantly more expansion occurred than was expected. As mentioned above, this could alert management to some future problems.

Unlike the other methods, a robust prediction strategy is not currently feasible. Although the strategy may be viable for specific cases, more research is required into predicting future expansion of software projects. In particular, it is not clear to what extent the current results may be automated or generalised. Also, although the overall increase in size can be predicted, more detailed predictions are still required, such as changes in size at the module level, or where in the project new modules are likely to appear.

3.4.2. Summary

As hinted at in the previous section, the most successful implementation of a future proof visualisation would include a combination of most, or all, of the above strategies. A big-box strategy will work much better if abstraction is used to minimise the amount of new information that will be added. Adding animation to this process, for example, when the box is filled, will also maintain or break context as required. Prediction can then be used to gradually resize areas within the box. As long as the movement is sufficiently gradual, it will not prove to be distracting.

No matter what strategy is employed, at some point there will be data changes that can not be accommodated easily within the visualisation. A line has to be drawn as to when the current view should be manipulated, and when a completely fresh view should be generated. However, if it is necessary to create this new view, it is not appropriate to simply switch from the old view to the new one. Rather, once the new view is generated, the visualisation should be run from an earlier time within the data, to show the new view being built up. This may allow some degree of context to be maintained as the user may recognise some of the changes that occurred and will be able to map, to some extent, the old view with the new one.

3.5. Metaphor

Research has shown that the use of suitable metaphors is beneficial for improving task performance [Dutton99]. The purpose of metaphor within visualisation is to reduce the cognitive overhead in interpreting an image. This is achieved by ensuring that a representation is used that emulates a familiar

physical environment. The structure and relationships within the environment may then be used to represent the structure and relationships within the data. For example, a tower block may consist of a number of floors. Therefore, a 'contains' relationship within the data may therefore be shown using the tower block. By doing so, the user does not have to recall that a floor within a tower block represents that relationship, and instead the relationship is made subconsciously. Therefore, the user should realise that the data represented by the floors and tower block also has that same relationship.

The most popular real-world metaphor currently used within software visualisation is a city metaphor [Charters02, Eick02, Knight99, Santos00, Young99]. The cityscape is usually used as a container for the elements of the visualisation. This cityscape can be split into districts, representing projects or modules. Within the districts, buildings are used to represent the actual object under consideration. The size of the building will often map to the size property of the object – LOC etc. The building will also be textured or coloured to show other properties, such as the number of methods if the building represents a class, or the type of access, number of parameters etc. if the building shows a method.

3.5.1. Issues

There are some obvious extensions to a static city metaphor for a dynamic visualisation. For example, roads could be used within the city to represent data flow, with traffic density along those roads indicating the frequency and amount of data that is passed between classes. Similarly, lights could be turned on or off in buildings depending on the last time they were called.

However, evolutionary visualisations – where additional data will be presented as the underlying project evolves – map less well to a city metaphor. At a first glance, there are some attractive options, for example, as represented in Table 3-1 below:

Underlying project change	**Representation within metaphor**
New class created	New building created
Class modified	Scaffolding appears around the relevant building
Class deleted	Building destroyed
Method added	New floor added to building
Method deleted	Floor removed from building
Increase in class LOC	Buildings increase in size
Increased use of class	Lights within building on more frequently.
Recent modifications in class	Clean paint work
No recent modifications	Paint work begins to peel

Table 3-1.Possible visual representations to highlight evolution.

Watching the city evolve would be particularly informative. For example, a relatively unused area within the city might suddenly light up, as new functionality, which was previously developed over a period of time, was brought into use. This may be as a result of a change in the user interface, or within the code itself.

However, the first six of these mappings will also break a conventional city metaphor. The power of a metaphor comes from the ability of the user to be able to map the metaphor to the data as easily as possible. Therefore, unrealistic operations occurring within the real-world view will suggest that the underlying project is also undergoing some unusual operations.

For example, creating a new building either requires empty space in the city, either in the centre or on the outskirts, or destroying a building to create the new one. It is not acceptable for buildings to be pushed to the side to make space for a new one, as this is not a valid real world operation. Similarly, removing a floor from a building is acceptable if the floor being added is at the top of the building. However, if the floor occurs half way down, then destroying just that floor, with everything else dropping down to fill the space, would not map to a real world situation. However, metaphorically speaking, these are precisely the types of operations that occur ordinarily within evolution, as classes and methods are created and deleted. The user would then be alerted unnecessarily to what constitutes usual behaviour within the project, as it would be unusual within the metaphor.

Furthermore, although there has been much recent research into evolution and evolvability as described in section 2.2, it is still not clear how 'correct' evolution should occur. For example, is it better to make a small modification to a core class within the project to provide new functionality, or should many more modifications be made elsewhere to allow this functionality without affecting the stable parts of the system? Such open questions have a big impact on the usefulness of metaphor within evolutionary visualisations. If modifying a core class was found to be a bad operation, then it would be appropriate to have a real world mapping where the entire building had to be demolished and rebuilt to accommodate the change. However, if modifying a core class was preferred, then a much less drastic representation would have to be found. Similarly, if creating new modules within the core part of the project was found to cause many problems, then space could be created within the city with earthquakes, or by creating a volcano within the central part of the city. Such operations would be consistent with the metaphor, and would instantly alert the user of the visualisation that there were serious problems in that area.

3.5.2. Summary

Although metaphor may be used to represent aspects of software, the fact is that software is virtual rather than physical. Therefore, changes may be made to the software that would be impossible in a physical environment. Modifications mean functionality may be added or deleted anywhere within the entire project, with no need to consider the laws of physics, or the resources that would be necessary in a real world environment. Therefore, without an existing physical environment that allows these

modifications to be supported, it is not feasible for an evolutionary visualisation to be able to implement a metaphor completely, and so fully achieve the benefits offered. Failure to implement the metaphor completely results in the misunderstandings and false alarms presented earlier. Therefore, until software evolution is better understood, it is necessary to use an abstract representation when developing evolutionary visualisations.

3.6. Animation

Animation may be defined as:

> *"The technique of filming successive drawings or positions of puppets or models to create an illusion of movement when the film is shown as a sequence."* [Allen90]

This definition may easily be expanded to consider computer animation, where no film is required and images may be generated as necessary and displayed on screen. However, the definition also needs to be extended within the context of visualisation. Animation can be used to achieve one of two distinct effects.

The first of these is when the viewpoint, or camera, into the visualisation is changed. A smooth zoom operation is achieved by slowly moving the camera into the image, and displaying each of the images captured. The result qualifies as animation, as the user perceives movement of the camera. Similar camera movements can achieve scrolling, and rotation. More complex operations, such as a transition to a fish-eye view can be achieved by conceptually inserting a number of lenses of various strengths in front of the camera. Similarly, the whole image can be darkened or tinted by applying suitable filters in front of the camera. Although this form of animation is vital within visualisation, the application is almost always an implementation issue. Therefore issues regarding this will not be addressed in this section. Where necessary, this form will be referred to as a '*smooth camera transition*'.

The second is where objects or groups of objects within the visualisation need to change their location or representation relative to the old or surrounding objects, in order to represent the underlying data set. This rules out scrolling and zooming, as the relative size or location of the objects remains the same. However, independent movement, such as a car moving along a road in a city-based visualisation, or a building growing slowly, is included within the definition. It is this form of animation that will be addressed, and references to '*animation*' refer to this form.

With this form of animation, application is no longer just an implementation issue. Therefore, some issues regarding the use of animation within software visualisation will be examined.

3.6.1. Three dimensions

Although many of the available software visualisations were developed for 2D, there are now an increasing number of 3D visualisations that exist. Reasons for this progression are numerous. The main issues are that the addition of a third dimension creates far more space to lay out the data, and that the extra dimension means that more properties about the data may be shown simultaneously. Other

reasons are that a 3D representation is more suitable for a real-world metaphor, and 3D is often considered to be more aesthetically pleasing.

Applying animation within a 2D visualisation will also result in a three-dimensional visualisation, with time considered as a dimension. Therefore, it is important to examine the relationship between a *3D* (x,y,z) visualisation, and a *three-dimensional* (x,y,time) visualisation, and the benefits of each.

It is always possible to map from a 3D visualisation to a three-dimensional one. Indeed, the process is a vital one with current technology. Viewing a 3D visualisation through a 2D display requires a 2D image to be created. Perception of a 3D world is only properly achieved when the 2D image changes to show the 3D world from a different viewpoint. Therefore, smooth camera transitions in the 3D world, when mapped into 2D, will result in a three-dimensional visualisation.

Similarly, mapping from a three-dimensional visualisation to 3D is also possible. By mapping time to the third axis, solid shapes will be built up from the animated 2D view which are then represented in a 3D world. An example of this relationship is shown in Figure 3-4

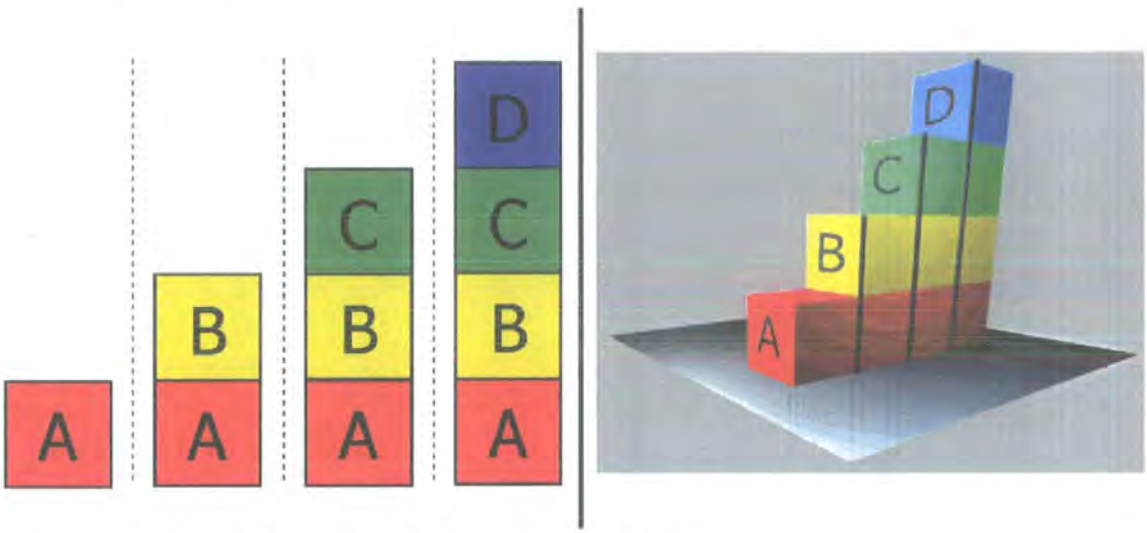


Figure 3-4. Four frames of animation and an equivalent 3D representation.

Given this close relationship, are there benefits of using 3D over three-dimensional, or vice versa? At present, this is difficult to answer. Many current 3D visualisations do not significantly utilise all three available dimensions, and a 2D representation is more than adequate, making a fair comparison difficult. For example, a city-based visualisation will often lay out the city on a horizontal plane, with the height of the building representing some property of the underlying object. Although all three dimensions are used, the height of the building is the only vertical property. Within a real environment, buildings would be placed at different heights depending on the landscape, with hills and valleys possibly representing other properties. Furthermore, objects can float in space, or be buried underground. Similarly, there are many graph-based visualisations that draw the graph in 3D rather than 2D. Here, the third dimension is not used to represent any property, but used to gain more space.

Many current 3D visualisations can be modified to work in 2D with a very small change in representation, and no loss of information. For example, it may be possible to map the 2D plane into a

1D list. In this case, each element in the list can be sized to represent the building, with the building height mapped onto the height of the element. Figure 3-5 demonstrates a view of one aspect of Software World shown in Figure 2-3 but in 2D. A coloured bar at the base of each building is used to signify the block that the building may be found in. Alternatively, the 2D plane can be retained, with some other property representing height, such as a single line at the correct position. This is demonstrated by Figure 3-6, where a plan-like view of the same Software World view is shown. In this case, a green bar to the right of each building represents the height. Although it can be argued that such a change reduces both the benefit of the metaphor used, and the impact of the building height, the benefit is that the true height of the building is shown regardless of perspective or whether it is partially obscured. Changing a graph laid out in 3D into one in 2D is often a trivial operation.

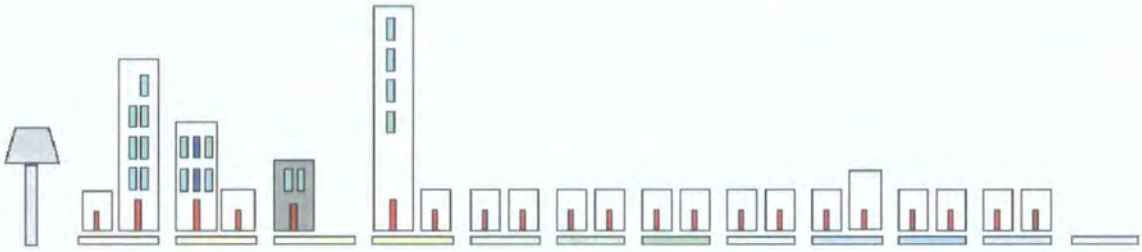


Figure 3-5. 2D equivalent of part of Software World, as a list.

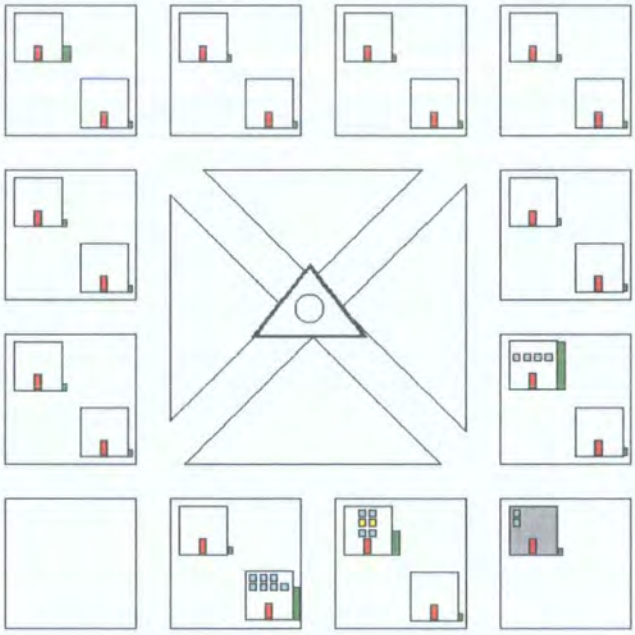


Figure 3-6. 2D equivalent of part of Software World, as a plan view.

There are, however, some 3D visualisations that do significantly utilise the third dimension, [Riva98, Chi94, Koike97], and these are appropriate for further analysis. Interestingly, in these cases, the third dimension is mapped on to time, or some time-dependent variable. Similarly, animated visualisations will usually show progress over time, such as in algorithm animation. One of the reasons for this is that

software code is naturally one-dimensional, or two-dimensional if indentation is considered. Most mappings into 3D are based on this 1D or 2D structure, and so a reverse mapping is usually possible. However, adding the concept of time introduces a genuine new dimension to the data, which makes the 3D mapping onto 2D much more difficult without losing information.

3.6.2. Overloading

Chapter 2 defined two main uses for animation within software visualisation. Firstly, animation may be used as a purely visual property, in order to attract the user's attention to a particular area of the display. Secondly, animation may be used to show additional attributes of the data. For example, allowing elements to spin within the visualisation allows attributes to be mapped onto the speed and direction of the rotation. Alternatively, the location or shape of an element may change during the animation, to reflect changes to the data over a period of time.

Although it is possible to combine these methods together with animation that is used to re-layout elements in a view, there is a significant danger of overloading. The above methods use animation to convey information about data, or highlight unusual or interesting areas within the visualisation. When modifying layouts, animation is used to maintain context, and provide a smooth transition. Therefore, it is important that these are separated. Otherwise, confusion will arise as to whether objects moving relate to the data, or whether the objects are simply moving to a different location for layout reasons.

There are at least two ways of solving this problem. The first is to contain all representation-based animation within an obvious container. Any animation occurring within the container should only refer to data. Movement of the container itself relates to layout issues. An additional advantage of this approach is that the relative location of the objects within the container remains the same when the container moves, making it significantly easier to maintain context, or even ignore the container movement altogether.

The second approach if this is not possible is to separate the animation into two or more distinct stages, as also recommended in section 3.4.1.4. During the data-animation stage, it may be the case that interaction is supported, and objects can be selected and manipulated. In this case, during the separate layout-animation stage this interaction should be disabled to indicate that layout changes are totally unrelated to the data. Alternatively a coloured border could appear during a layout change, again making an obvious difference between the two stages.

Whether 'smooth camera transitions' can be integrated directly will depend on the required operation. A simple operation such as zooming or panning could be integrated without causing any problems. Applying fish-eye lenses may make it more difficult for the user to view the data animations at the same time, as movement and distance will become distorted. In this case, it would be preferable to pause the data and layout animations whilst the lenses are applied.

3.6.3. Temporal issues

There are a number of other conceptual problems regarding the use of animation in visualisations. Firstly, unlike a static visualisation, animated objects are transient. Therefore, if the user is not looking at that object when it appears or disappears, there may be no indication that it was ever there. Within a 2D view, this is most likely to occur during a zoom operation rather than in normal use, as layout algorithms should be able to avoid the problem of overlapping nodes. If an animated 3D visualisation is used, creating a 4D visualisation of x, y, z and time, then occlusion is much more likely as the user will often have control of the camera. Therefore, objects are more likely to be obscured by others on the grounds that the user may navigate around them. One way of resolving this issue is to always have an overview window of the entire visualisation that highlights new and deleted information. Even if the user was not focusing on the area of activity, they would are at least be aware that something had happened.

Such an operation also requires bi-directional playback of the animation, ideally at different speeds. This allows a user that did witness some unusual activity in an overview window to zoom in on that area and view the animation again. Operations like this mean that pre-rendering the animation is unlikely to be possible, as the user must be able to interact with the visualisation at any point. This also raises the problem of multiple time-lines – a similar problem to 'forward' and 'back' controls within a web browser. By playing the animation forward, and then backward, and making some change at that point, playing the animation forward again will produce a different result to the first time. This may affect bookmarking operations, for example.

Finally, it is important that the animation is also useful when viewed statically. Obviously, the same richness of information can not be available, as otherwise adding animation would be meaningless. However, much of the exploration is likely to take place when the animation is paused, as movement will then not distract the user. Similarly, if a printout is taken of the visualisation, this should be relevant. Possible solutions to this may include reinforcing any changes that occur. For example, if a new node representing a new class appears during the animation phase, that node should be given a different border colour or some other property to indicate that it is new. As the animation continues to play, this border should then disappear.

3.6.4. Design issues

There are a number of examples of visualisation research where animation is suggested as a means of enhancing the visualisation in order to display historical information contained within the data set [Bartram97, Carr99, Knight00b]. However, despite this, new animated visualisations are very rare, and most research effort is concentrated on 3D visualisations instead. It is difficult to deduce exactly why this is the case, although there are a number of possible reasons.

Firstly, the difficulties with using animation are not always obvious. First impressions often indicate that it is possible to 'tack on' animation to an existing visualisation with few problems. As has been shown in this chapter, this is unlikely. An initial problem might be the vast increase in data that must be

handled by the visualisation. A visualisation that was implemented to handle 100KLOC will suddenly have to store 1MLOC if just 10 versions are presented. This may require the implementation of new data structures and algorithms to deal with the data efficiently. Graphically, the visualisation may have to be simplified so that a display that used to take several seconds to draw becomes a fraction of that, so that a reasonable frame rate can be achieved. Once the pure implementation issues have been resolved, the other problems of visualising evolving data are then introduced, such as those of familiarity and metaphor.

Secondly, it may be that visualising historical information is not seen as important. In the same way that software visualisation sometimes struggles to make an impact as the need is not recognised, historical information may suffer in the same way. Recent studies have gone some way to show the extent and relevance of data contained within a version control repository [Ball97], and so it is anticipated that more visualisations, whether 3D or three-dimensional, will start to be based on this.

Thirdly, there is poor tool support for creating animations of this nature. 3D visualisations can be generated in a number of suitable languages and tools. Prototypes can be generated relatively easily in tools such as 3D Max, or in modelling languages such as VRML [Carey99]. Implementations can be based on libraries such as Java3D [Java3D] or Maverik [Hubb96]. Animation tools of the same high standard are much more rare, of which Macromedia Flash [Flash03] is the best known. Although Flash is designed for creating 2D animations, implementation of these must be done in Flash's own scripting language, rather than a mainstream language such as Java or C++. Although there is some third party support for generating Flash files within programs, using tools such as Ming [Ming03], these are currently at an early stage and nowhere near as complete as the 3D equivalents. These files must then be run through a Flash viewer, making fully interactive systems more difficult. Flash was also designed as a web-based animation tool where there tend to be a few large objects at once, and so it is optimised for this situation. Visualisations will often be the opposite, with many small objects displayed simultaneously.

Finally, there are aesthetic benefits to 3D environments. As computer games have become more powerful and graphically complex, new hardware has been introduced to handle these new demands. Displays with many textured objects and lighting effects can be updated at exceptionally high frame rates. Much research concentrates on algorithms and structures that will further enhance these speeds. Therefore, graphically impressive environments can be displayed and updated in real time. Unfortunately, there is not the same support and focus for 2D animation. 2D animations will usually have to be graphically simpler than the 3D equivalent in order to achieve the same frame rates. Although good graphics are unlikely to have a serious long-term impact into the use of the visualisation, first impressions may increase the number of users that start to use the system.

3.7. Summary

This chapter has introduced the concept of evolutionary software visualisations. In particular, the differences between static and evolutionary visualisations have been identified. These differences

affect the design of a visualisation significantly, in terms of both the layout and representations that should be used. Specifically, the concept of 'future-proofing' as being vital to an evolutionary visualisation has been introduced, and the effect of supporting this concept within visualisations has been outlined. The difficulties of using a physical metaphor to represent virtual software that may evolve in an unconstrained manner have also been described.

As animation is a natural means of representing changes over time, the chapter finally examines the benefits and dangers of using animation within software visualisation. A number of warnings and guidelines have been identified as a result, in order to avoid some of the problems that animation may introduce.

The following two chapters introduce two new animated visualisations, that are based on the concepts and guidelines contained within this chapter.

ANIMATING THE EVOLUTION OF SOFTWARE

4. Revision Towers

4.1. Aim

As chapter 2 described, the use and development of open source software has increased significantly in recent years, with some positive results. One of the important aspects of open source development is the need for large numbers of developers to assist with the project. Almost all of these will be volunteers, and, importantly, they will unlikely to work on the project full time. The vast majority of contributions are likely to be bug reports, followed by fault fixes, and new code is responsible for the remainder.

A second aspect is that most, if not all, of the development will be distributed world wide. The predominant form of communication will be online forums or mailing lists, or, failing that, examination of the source code itself. Therefore, with this limited communication mechanism, it is unlikely that every participant will have a good understanding of the software project. Additionally, the communication mechanism is usually informal, and so the project will not have a defined communication structure as in a CSCW type project.

The situation is made more difficult with the large number of modifications made to the software on a regular basis. For a developer to make a successful modification to the software, it is important that they have an up to date mental model of that software. Frequent changes can make this difficult. In particular, if the developer is unable to work on the project, and returns some time later, there may be a large number of changes that are difficult to identify without working through all of the mailing list archives. Documentation of changes and their impact will obviously make this process easier, but access to this information can not be guaranteed within most projects.

Alternatively, it may be the case that a skilled software engineer wishes to join the project. As the documentation often falls behind the current state of the software, the engineer must rely on the source code to comprehend the system. However, this is a difficult task, with possibly hundreds or thousands of files of source code to consider. In addition, it may be difficult to determine which are the key files that are worth studying in detail.

Finally, consider the perspective of the manager of a project. It may be that they wish to understand the areas of the source code undergoing most development, and those that have not been recently worked on. Alternatively, they may wish to examine the reliability of code developed by a particular author in order to encourage them to work on a more, or less, critical part of the project. Finally, they may wish to see the parts of the project that become affected when new functionality is implemented.

That there is a need for additional support to help with these issues is also clear. Currently, tool support integrated into online open source repositories such as SourceForge is limited. Essentially, a web-based version of Diff will ordinarily be provided, allowing the changes between two versions of the file to be compared at any one time. Colour will usually be used rather than the default output to highlight changes. Also, it will also be possible to view the author responsible for a particular line of code. Diff-based tools have a number of problems however, many of which were examined in chapter 2.

With this in mind, this chapter presents the concepts of a visualisation, Revision Towers, which seeks to provide some solutions to the scenarios presented above. Revision Towers avoids many of the problems of Diff, as well as the expense and difficulties of syntax-based comparisons, by examining changes at the repository level. A significant amount of information is lost with this approach, such as being able to refer to changes in the control flow or complexity of the project. However, there is still more than enough data from other sources such as log files which include the time and size in lines of code of any changes made, and mailing lists which can provide the motivation for the change, to provide solutions to the above issues. This information is often exceptionally long and verbose, and spotting interesting patterns and trends through reading alone is almost impossible. Therefore, visualisation will be used to highlight areas of interest.

Two similar tools exist, as mentioned in chapter 2. VRCS is an attractive approach, but only suitable for very small repositories. It shows clearly when files were added to the repository, and changes made to them. It also supports interaction, allowing files to be checked out and compared easily. 3dSoftVis is more suited to large-scale repositories and presents an overview at a managerial level. However, in creating the abstraction mechanisms to cope with the scale required, much of the information interesting to a developer is lost. Revision Towers aims to address both issues by providing a representation that will handle larger data sets, without abstracting too much information.

In addition to these criteria, Revision Towers must satisfy a number of other constraints. The audience will be developers and project managers, with little or no prior experience of visualisation. This means that the representation must be intuitive, with a relatively low learning curve. Failure to achieve this would mean that the tool would be unlikely to be used, with developers resorting to their previous, less suitable, tools.

The second issue is that the representation used should support evolving data sets, as set out within chapter 3. As releases occur on a regular basis within open source software, it is highly unsatisfactory for the visualisation to present a completely separate picture to the user for every separate release. Therefore, some of the techniques for managing future-proofing will need to be incorporated.

Finally, the tool will ideally be placed alongside existing tools available within the repositories. As most of these tools are designed for output to a standard display, it is not possible to assume that specialised interfaces will be available for interacting with it. The visualisation is therefore limited to a standard display, using mouse and keyboard for interaction.

Four aspects of the Revision Towers visualisation will now be presented. Firstly, the representation and layout algorithms used will be examined. This will be followed by looking at the role that animation plays within the visualisation. Finally, the available interaction mechanisms will be described. The following descriptions assume that at least the following repository data is accessible for the visualisation: the names of the files within the project, the dates of individual versions and releases relating to those files, the author responsible for those versions, and some form of comment field describing the version. In addition, the forms of raw data that are available within a repository will be examined, to determine whether the approach is feasible.

4.2. Representation

The central concept within the visualisation is a 'tower'. A tower represents two versioned, related files that are viewed side by side. The central section represents software releases, with the earliest releases at the base of the tower, and the latest, as yet unreleased, at the very top. Each side section represents the history of a file, and shows how the individual versions of a file map to the releases, as shown in Figure 4-1. This shows the roles of three authors (A, B and C), and how they have been involved with specific versions over a period of 11 releases.

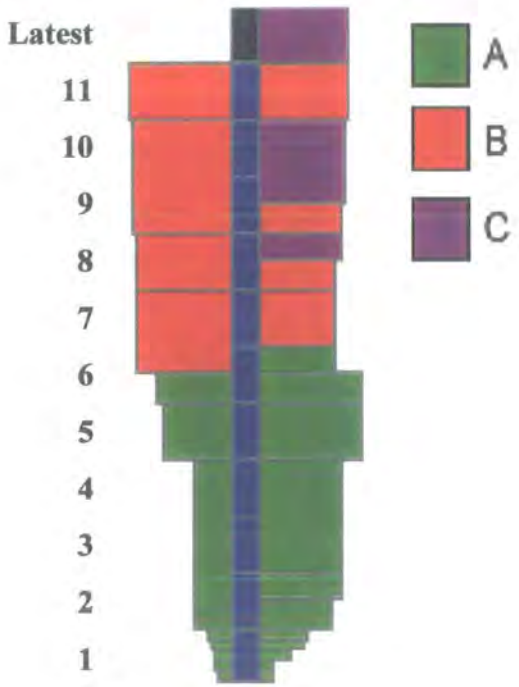


Figure 4-1. A Revision Tower.

The main intention of pairing files is to show the relationship between the interface and implementation of a module. For example, in C or C++, header files (.h) and implementation files (.c) may be compared against each other. A header file will always be shown on the left side of a tower, and the implementation file on the right, to emphasise the differences between the two types of file.

Although the pairing may take place automatically taking advantage of standard naming conventions, the user may also control the pairing process. In addition, the same file may appear in multiple towers. This is desirable if the implementation of a class covers more than one file, or if many classes are contained within a single header file.

The towers are then displayed in a grid formation to fill the available display area, and are ordered according to the date that the files were created.

4.2.1. Filename

The filename is not displayed permanently, in order to allow more space for the remainder of the view. However, it may be displayed at the top of each tower. Assuming that the header and implementation files are named identically, then this is the only name shown. Otherwise, filenames may also be displayed over each side of the tower. The filename is displayed with an additional short identifier, with files from the same directory given the same identifier.

4.2.2. Releases

Each tower is initially normalised to be the same height. Within this, all releases of the software are shown. Therefore, a file introduced at a later date to the project will have no versions associated with the early releases, as shown in Figure 4-2. In this case, the base may optionally be compressed to reduce the space required, as shown in the right of the two towers.

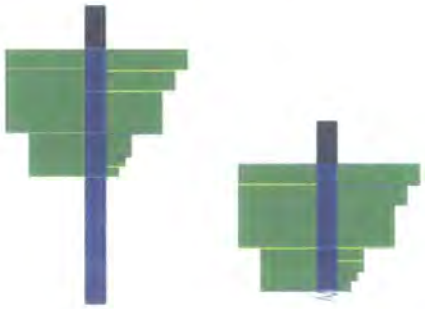


Figure 4-2. Two towers showing the representations for a file introduced at a later date.

The height of a release is determined in one of two ways, depending on the preference of the user of the visualisation. Each individual tower may have a different allocation mechanism.

The initial view is to provide each release with an equal proportion of the height of the tower. Each release is given the same importance as any other release, regardless of the time of that release. This approach is particularly useful with smaller projects. As there are fewer developers working on the project, none of whom are likely to work full time, it is probable that there will be occasions when they will have no time to spend on development. This leads to very irregular releases, with the time between releases based not on the complexity or the number of new features, but rather the amount of free time the developers had. By allocating each release with an equal amount of space, this problem is ignored.

A second view allows the size of the segments to represent the length of time of each release. This will be more relevant for larger projects, with a more definite release schedule. Whereas ten file updates within a single release may appear intensive with the default view, if this release is shown to have taken ten times longer than the average release, a more accurate picture is obtained. Conversely, a small release immediately after a large '.0' release will have a proportionate height, and so will not indicate that little work was done over a larger period of time.

4.2.3. Versions

Each side segment of the tower represents a specific version of the file. The side segments have variable widths, and are used to show the change in file size. The change in file size, rather than the overall length of the file, is shown by default. However, if the initial length of the file is known, then the number of lines in the file may be shown instead.

The width of the segment is determined relative to the maximum number of changes within that file. This avoids the difficulties of changes to small files being invisible due to being swamped by much larger files. A coloured line at the base of each tower is used to display the multiplying factor involved, so that a large file will have a longer, brighter line than a small file. This is not intended to allow meaningful comparisons, but rather to alert the user to the fact that there is a difference in file size.

The user may then decide how to display the width of the segments, according to whether they wish to show relative sizes, absolute sizes, or a semi-relative view where header files and implementation files are considered separately. This last view is important, as header files will usually be much smaller than implementation files. By providing this composite view, information in the header files is also clearly visible. These different allocations are shown in Figure 4-3. As can be seen, the semi-relative view maximises the available display space. Alternatively, although large files affect the absolute size view, it represents the true size of the individual versions.

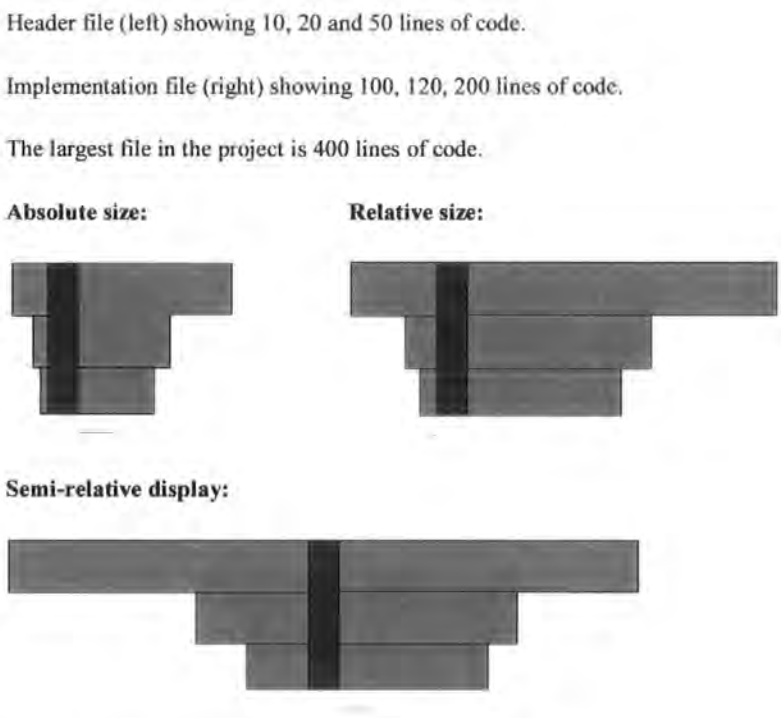


Figure 4-3. Three different width allocations for the same data.

The height of the side segment may be set in one of two ways. The heights may be allocated within the space of that release, for example, a file undergoing two revisions within one release would mean each side segment had a height that was half of the height allocated to the central release segment. This view is useful as it gives a clear picture of how often each file changes per release, or within the project as a whole. The danger of this approach is that comparisons between the header and implementation file within a tower may become meaningless. Instead, comparisons should be made during the animation process described later.

Alternatively, the side segment heights can be allocated proportionally to the time of check-in relative to the release dates, so a version checked in very early after the previous release, and well before the next one would have a short height. This is a similar process to the resizing of the central segment. This is the most useful view when comparing check-in dates without using animation, for example, with a print out of the display. The danger of this second approach is that the times are relative to the length of the release. If all releases have been set to be the same size, which is the default behaviour, then two versions with the same height will not have taken the same length of time.

By combining the two mechanisms for determining the height of releases and versions, four different towers may be generated for the same data, as shown in Figure 4-4. Here, each tower is shows 14 versions of a file, spread across four releases. The central segment is coloured accordingly to indicate which allocation method has been used.

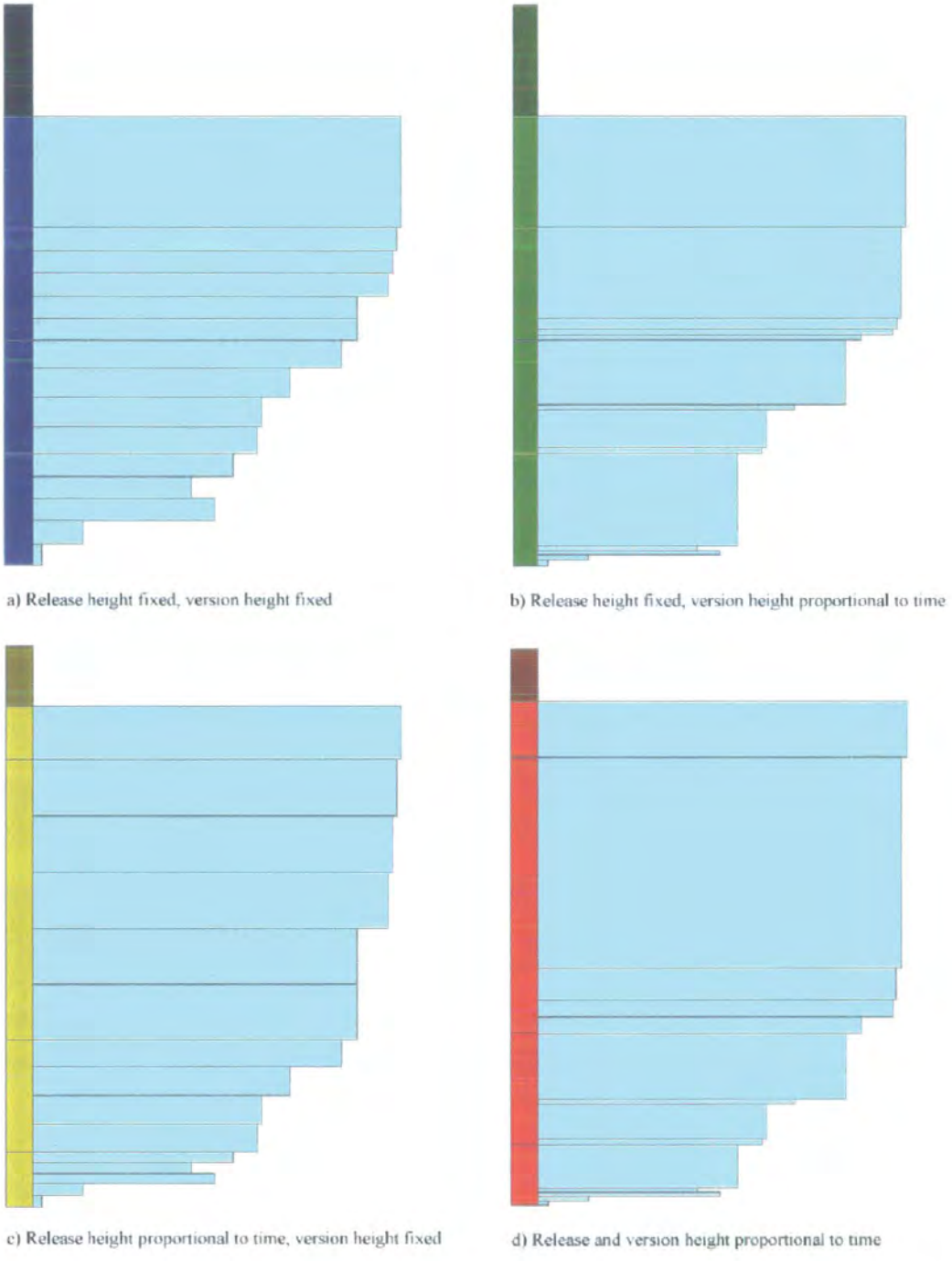
Finally, a change indicator may be optionally added to the outside edge of each segment, in the centre. The purpose of this is as an aid when the zoom level is such that the number of pixels available to show a segment is too small to show any difference in file size. In this case, two segments will be shown with the same width, even though one may be longer than the other. The purpose of the indicator is to indicate whether the file has increased or decreased in size. If the file has increased in size, the line is added to the outer edge. If the file size has decreased, the line is added to the inner edge. Although small, the difference is enough to alert the user that the size has changed. Further details can be obtained through other means within the visualisation. The indicator is shown in Figure 4-5, and indicates that all versions have grown in size, with the exception of the top left version which has reduced slightly.

4.2.4. Authors

Each side segment may be rendered with a different colour providing further information about the specific version, such as the author responsible for the version. In this case, each side segment is filled with a colour that maps to a legend representing all of the authors. Importantly, these colours are generated consistently, by allocating colours to authors in order of the time that they were first involved with the project. Therefore, the author who submitted the first file will be allocated the first colour, and this colour will remain throughout the lifetime of the project.

Once the initial colours are allocated, the user of the visualisation can then change these to ones that are more easily recognised. In particular, several authors can be allocated the same colour, allowing better

recognition of groups. Uninteresting authors can also have no colour allocated, which has the effect of highlighting the other authors.



a) Release height fixed, version height fixed

b) Release height fixed, version height proportional to time

c) Release height proportional to time, version height fixed

d) Release and version height proportional to time

Figure 4-4. Four height allocation algorithms.

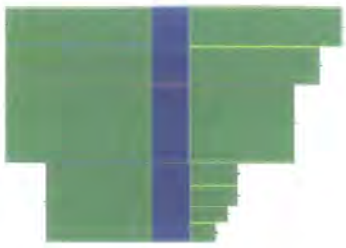


Figure 4-5. Part of a Revision Tower with added change indicators.

4.2.5. Comments

Revision Towers supports the inclusion of a limited amount of the type of data that may be contained within comment fields. For example, if the author of a change is not the same as the user who submitted the file, then the original author may be contained within this field. In this case, the user may toggle the author display to show details from either source. If two or more developers contributed to a single version, the segment will be split vertically into the necessary number of parts with each part coloured individually. As with the dedicated author field, colours will be allocated to ensure that the same developer is always allocated the same colour.

An alternative piece of information from the comment field that may be displayed is the type of maintenance that was carried out within that version. This may be derived from the types of words used to describe the changes made to the file for that version [Mockus00a]. In order to display this information, a small shape is displayed in the centre of the segment, as shown in Figure 4-6. If there is no space within the segment to display this information, or the type of change could not be identified, then the shape is not shown. The colour of the shape is either the background colour of the display, or the colour of the border of the tower, depending on which is more visible against the segment colour. Alternatively, the whole of the segment colour can be assigned to the maintenance type instead of showing the author information, which may be more useful when the display is zoomed out.

The shape technique can also be used if other information is provided. For example, by linking versions to fault reports, the severity of a fix may be viewed. A file with many severe faults associated with it would be a good candidate for re-engineering.

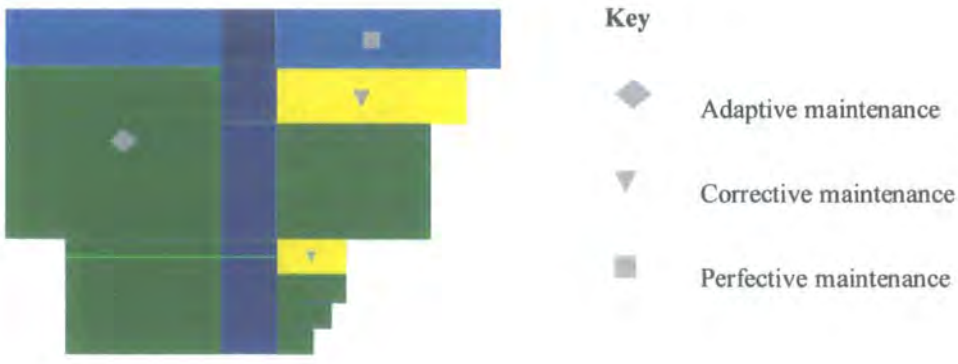


Figure 4-6 Section of a tower showing three authors and the type of maintenance activity identified.

4.2.6. Branches

Branching of a file is common within large projects under configuration management. The purpose is often to allow experimental code to be developed, without affecting the stable behaviour of the code. Once the experimental code has been stabilised, it is then merged back into the main 'trunk'.

Revision Towers represents this in a simple fashion. A version of a file that has branches leading off it is shown with an arrow leading out from the edge of the segment representing that version. The presence of branches may often be determined easily.

In some circumstances, it is also possible to determine that code has been transferred from a branch to the stable trunk. An arrow leading into the segment will show that a merge may have taken place. The notation does not indicate the source of the merge, although this may be provided if further details are requested. An example of this is shown in Figure 4-7.

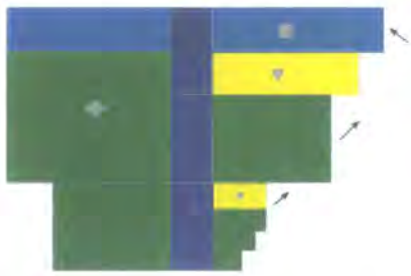


Figure 4-7. Indication of two new branches, and a later merge operation.

4.2.7. Multiple towers

The previous explanations have referred to the use of a single tower, representing two files. However, further issues are involved when many towers are considered, each representing a different pair of files.

In order to provide a consistent picture of the entire project, the central section of each tower is identical, with space allocated for every release that has been made as part of the project. If the files shown by a tower are not part of a particular release, then no version information will be shown for that release. For example, this is the case for files that join the project at a later date, as shown in Figure 4-2.

In some cases, the releases will not be associated with a version on the trunk, but rather with a version on a branch. Where this happens, the release will be displayed on the main trunk as dots, rather than solid colour as would normally be the case. If the user wishes to view the release, it will be necessary to expand the branch. This is demonstrated below in Figure 4-8. Here, the second release shown is made up of a header file that is part of the trunk, and an implementation file that is contained within a branch.

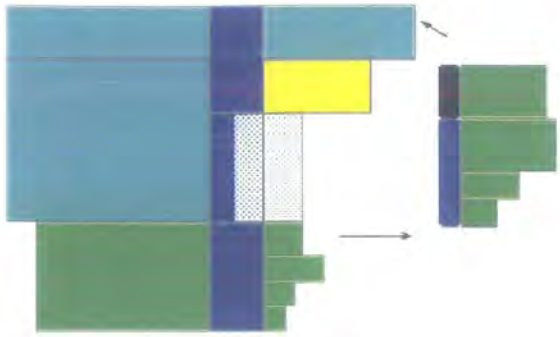


Figure 4-8. Expansion of a branch containing a release.

4.3. Layout

The layout algorithm used by Revision Towers is designed to support the future-proofing requirements specified within the previous chapter. As the visualisation is based on a regularly changing data set, it is inappropriate to use a layout algorithm that responds badly to change.

The initial layout method used is based on a big-box strategy. Towers are ordered chronologically, and then laid out in a grid from left to right, and top to bottom. Using the creation date of the file for the layout has a number of benefits. Firstly, the display will always appear consistent, with the top left tower always being the same. Secondly, the space allocation is efficient, with all of the unallocated space clustered at the bottom of the display, where new files included in the project are shown. Finally, each row of the display can have a different height. As new files will not be involved in early releases of the project, it is not necessary to display all of the releases within the central section. Therefore, if all of the towers in one row of the display missed an early release, then the compressed representation shown in Figure 4-2 may be used for all of these towers.

A small amount of space is left unallocated at the right side of the screen. The purpose of this is to provide some space for expansion of any branches occurring within towers. Space is also left unallocated at the bottom of the screen. This is provided as the 'big-box' aspect of the display, allowing towers to be added in the future. The space also doubles as an initial placement for a floating work area for the user, allowing them to copy partial or complete towers into it. This allows further comparisons to be made.

As discussed in chapter 3, the most significant problem of a big-box layout strategy is the behaviour when the box is full. If the visualisation is used, and then used again three months later with an extended data set representing the new data within that time, ideally a similar picture will be displayed to the user. Revision Towers allows two different approaches to solve this problem, of which either can be selected interchangeably.

The first is that the box has an infinite height, and so the box can never be filled. Practically, this means that a scroll bar is introduced, and that all of the towers that exist will not fit on one screen. This approach is suitable if the user is less interested in the initial layout, and is likely to regroup the towers anyway. The disadvantage is that scrolling is required in order to view the whole project.

The second option aims to keep all of the towers displayed on one screen, using abstraction techniques if necessary. The width and height of the towers determines how many towers can be displayed on screen at once, and is initially set so that the towers displayed make good use of the available screen space. By reducing the height of each tower, it is possible to increase the number that can be shown, whilst keeping the relative position of each tower the same. This is not the case with reducing width, however, as either space will be left, or towers will unwrap onto the previous row, and the relative location may change significantly. Therefore, when no more towers can be added by reducing the height of a tower sensibly, it is necessary to layout the towers again in a more efficient way. Animation is used to alert the user that this is happening. The purpose is not to allow the user to view the old and new location of every tower, and so maintain context, as this is unlikely to be achievable. Rather, the intention is to ensure that the user is aware that significant layout changes have occurred, and be aware that previous assumptions based on location are now unfounded.

A spiral layout would be significantly more forgiving in terms of reducing the width and height of a tower whilst maintaining the same relative layout. However, this was rejected for two reasons. Firstly, a spiral ordering is much less natural than reading left to right down the display. Secondly, the use of a spiral would be more wasteful of space, as every row in the display would have to be the same height rather than benefiting from the compression notation as shown in Figure 4-2.

4.4. Animation

Revision Towers uses animation for two separate purposes. Firstly, animation is used to show another dimension of the data. Secondly, animation is used to smooth transitions during any layout changes that are required, either for the whole display, or inside a single tower.

Time is the most natural value to map when using animation, and this is the case in Revision Towers. This allows the user to view the growth of the project over time. The basic method involves complete towers that are set up at the start of the animation, and then hidden. As the animation plays, the necessary parts of the towers are revealed. Within Revision Towers, the central structure is always shown in order to provide a central reference point. However, versions fade in when the virtual time within the animation maps on to the version check-in date.

The purpose of fading is to allow the user to see what is about to change. If the new data appeared without fading-in, and the user blinked while the new data appeared, change blindness means that they may not even notice that the picture had changed. The fade in process takes place over a short period of time, again to avoid change blindness. In addition, the good colour contrast against a black background means that changes are spotted more easily.

In order to provide context, a timeline is displayed during the animation. This timeline is scaled initially to represent the entire project. Releases are marked as long vertical lines, and are always mapped according to the release time, rather than equidistantly. The time of a release is not part of the log file, but may be extracted from mailing lists or change logs. The current point within the animation is shown as a long yellow bar.

The importance and content of a release is also shown. A white bar is used to represent a '1.0' release, which usually involves a large number of new features, or substantial reengineering of the code. A green bar shows a feature release, usually marked as '1.*n*'. A red bar shows a bug fix release, marked as '1.0.*n*'. A pale blue bar shows all other releases. By providing the visualisation with the format of typical release names, this information can be derived automatically. As the scale of the timeline may be such that every release can not be displayed, the spaces between releases in the timeline is coloured in repeating shades of grey. The purpose of this is that it allows the user to determine if many releases took place in a short period of time, as apparently adjacent releases will be coloured out of sequence. For example, Figure 4-9 shows a timeline with six releases – 1.0, 1.0.1, 1.1, 1.1.1, 1.2 and 2.0, approximately half way through the animation.

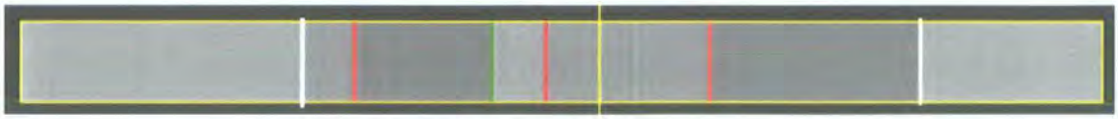


Figure 4-9. Example timeline.

Importantly, the user has full control over the animation. It is not sufficient to allow the user to just play the animation in a forward direction. If the user notices an interesting item within the visualisation, it is likely that they will wish to know where it came from. If only forward direction is provided, then they will have to reset the animation and monitor that point carefully. In turn, that may raise another interesting issue, where the whole process must be repeated again. Allowing bi-directional playback avoids this situation. In addition, several speeds are provided, allowing the user to get an overview in a short space of time, and then concentrate more on a specific area. Finally, the user may jump directly to any point in the animation, by selecting a point within the timeline.

When the user is allowed to interact with the visualisation during playback, the animation process becomes more difficult. The interaction features within Revision Towers are explained more fully in the following section. As an example however, consider expanding a branch within a tower. The effect of this is to show a new tower representing the branch at the side of the selected tower. Neighbouring towers are also moved in order to provide space. This operation affects the future of the animation, as some of the objects have now moved to a new position. Crucially though, it has also affected the *history* of the animation. If the user then plays the animation backwards, this must also take account of the fact that objects have moved to new positions. The process must also appear seamless to the user.

4.5. Interaction

Revision Towers is not just a static picture, but allows a large amount of interaction in order for the user to explore the data. The representation supports a number of different techniques.

4.5.1. Zooming and distortion

Although the initial display gives an equal prominence to each release within each file in the project, it is probable that users will be more interested in subsets of these. In addition, it is likely that the user will wish to examine some towers in more detail in order to view every version clearly. Zooming techniques are used to handle both of these cases.

In the first case, a technique is used based on the table-lens visualisation [Rao94]. As described, the central section of the tower is used to display release information, with space allocated equally or chronologically. However, if the first arrangement is chosen, the user can then increase or decrease the space allocated to a particular release. Neighbouring releases will be given a small increase in space, and distant releases will be compressed. In order to highlight this, the central section will change to a deep purple. In order to maintain a correct view of the mapping for the user, a time-based central section must first be changed to an equally spaced central section. This distortion effect can be applied to multiple releases and across several towers simultaneously. Animation is used to provide a smooth transition between the two views. An example of this operation is shown in Figure 4-10. The original tower is shown on the left, and the result of the table lens operation applied to the second release is shown on the right, allowing the versions within that release to be shown more clearly.

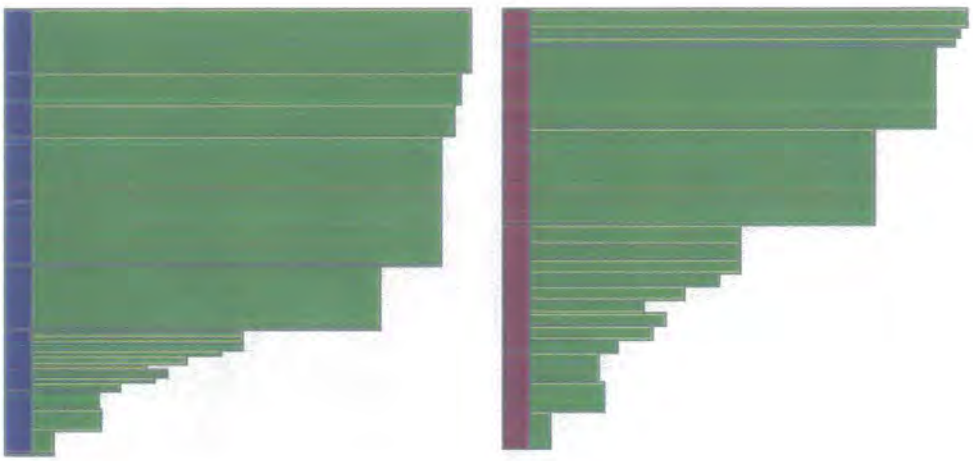


Figure 4-10. Result of lens operation.

A conventional zoom facility is also provided that allows the user to zoom smoothly, regardless of any other animated activity. Zooming will also consider level of detail aspects. For example, the initial layout algorithm used may mean that it is not possible to display every version, or every release, of a file. Therefore, abstraction mechanisms are used. If too many releases or versions are allocated to an area, then all of the changes from those versions will be grouped together. If versions have been grouped, then a single version will be shown representing the largest file size, and the most dominant author. The segment will be given a thick outside edge, with the thickness of the edge representing the number of grouped versions. If releases have also been grouped, then this will also be shown with a thicker border. In addition, a brighter colour will be used to colour the relevant central segment. The result of zooming out from Figure 4-10 is shown as Figure 4-11. Note the thick outside edge shown for

the versions in the second release, and the bright central segment at the top of the right tower indicating that releases were grouped.

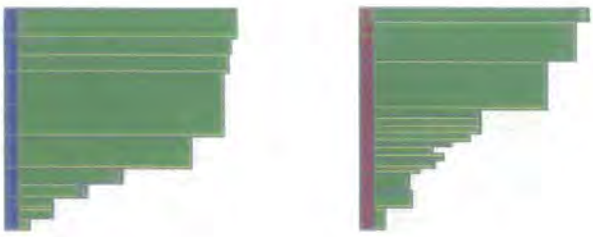


Figure 4-11. Result of zooming out from Figure 4-10.

It is also possible to zoom in and out of the timeline. This essentially has the effect of changing the overall speed of the animation within the visualisation. However, it also allows releases marked on the timeline to be seen more clearly.

4.5.2. Selection

Initially, each part of a tower is displayed with a dark grey border when first drawn. A selected object within this tower is then shown with a lighter border, allowing easy identification. Selections can be altered whilst the animation is playing.

Brushing is used to link views. Moving the mouse over a release in one tower will highlight that release in all other towers. If the release is currently compressed in another tower, then part of the border of that release will be highlighted. If the release exists within an unexpanded branch, then the branch arrow will be highlighted. The selected release will also be highlighted on the timeline.

Towers may also be mapped onto the timeline, providing a different viewpoint. Up to 4 towers may be mapped in this way. The check-in times of the versions within those towers are displayed as small coloured lines. Moving the mouse over these lines will highlight the versions within a tower, and vice versa. Moving the mouse over a tower will also map that tower temporarily onto the timeline.

A mouse over operation on any element within the visualisation will display the full details within a tool tip. This tool tip can be permanent if necessary, to allow comparisons of several elements simultaneously. The tip is shown with a semitransparent background to reduce occlusion. This allows direct access to the underlying data entities, such as the actual text contained within a comment field.

A search operation is also provided, intended to allow access to the comment fields. Searching for a particular bug number may show that several files across the project were marked with this number, giving some idea of the extent of the fix. Similarly, a fix may be spread across several versions in a tower, either indicating that it was a difficult problem to fix, or that it reappeared. It is also possible to show versions that have the same check-in date, or have the same comment attached, indicating that they were modified together.

A number of queries may be built up simultaneously. A version associated with a query is highlighted as normal. However, the border colour will also optionally pulse between grey and white. By observing the pulse phase and frequency, it is possible to determine which query, or queries, are satisfied.

4.5.3. Filtering

Filtering is used to reduce the amount of information displayed on screen. Revision Towers supports filtering of releases, and entire towers. As well as just compressing releases, the user may remove them completely from a tower, allowing the other releases to be expanded. A small gap is left in the central structure to indicate that one or more releases have been removed, and the version information associated with those releases is replaced with a single line. If this is done for every tower, then the release bar shown on the timeline is displayed as a dotted line to indicate this.

Towers may also be filtered. In this case, the entire tower is reduced in width and replaced by a single grey vertical line. Neighbouring towers may then widen to fill the space created. Queries will not include the tower in the results, unless specifically requested.

4.5.4. Integration

Revision Towers is not designed just to be a stand-alone visualisation. By providing clear access to the contents of the repository, it allows direct access to specific versions within specific files. This allows a number of typical operations to take place. By selecting a particular release or version, that release can be checked out in full. Similarly, selecting two versions allows the textual differences between those two versions to be shown. Finally, selecting two points on the timeline will allow access to the mailing list archives between those two dates, allowing the user to verify any findings made within the visualisation or to gain an alternative point of view.

4.6. Available Data

In order to demonstrate that such a tool is feasible, it is necessary to examine the data that is available without resorting to examining the source code. Within a typical environment such as SourceForge, there are a number of data sources provided that may be relevant.

4.6.1. CVS

CVS is almost synonymous with open source development. Therefore, it is certain that data available from CVS will be applicable to most open source projects, and so relevant for a visualisation such as Revision Towers. Information can be retrieved for individual files, or for the entire project, by using the log option available. This log file contains the following useful information for each file:

Working file, RCS file: Directory and filename of the selected file. The working file contains the name within the repository, whilst the RCS file gives the name and path when checked out.

Head: The number of the latest version of the file to be checked in.

Symbolic names: A series of entries tying a symbolic name (e.g. RELEASE_1) to a revision number (e.g. 1.13). Symbolic names are used to enable easy access to a particular versioned set of files later. For example, marking the latest state of the project as 'RELEASE_1' will mark the latest versions of each file within the project with that name. These can then be accessed at a later point in time.

Description: A series of entries, one for each time the file was checked in.

The log file also contains a series of entries, one for each time the file was checked in. Each entry is indexed by revision number, and also includes:

Date, Time: The date and time at which the file was checked back into the repository after modifications.

Author: The name of the developer, or manager, that checked the file in. If the developer making the change did not have write access to the repository, the author is likely to be the 'owner' of the file or module. Otherwise, the author is likely to be a trusted developer responsible for the modification.

Lines: The number of lines changed as a result of the modification, represented by *+new*, *-deleted*. This is taken straight from diff, and so +1, -1 may mean one new line, one deleted. However, it may also mean that just one line was edited instead.

Branches: Lists the latest version number at the end of a branch based on this version.

Comment: A free-text field, where a comment connected to the modification should be attached. Many projects will impose a fixed format on this field, requiring details such as the developer creating the patch, or a fault report or bug number that the patch is intended to fix.

From this, further information can also be deduced. For example, two files that were submitted to the repository with the same author and timestamp are likely to have been modified together, and so may be related. Similarly, as a single comment is requested when multiple files are checked in, identical comments in multiple files might also signify related files. This information is obviously not as accurate as configuration management systems that contain this information explicitly.

Parsing the comment field may also prove useful. In particular, searching for words such as 'optimise', 'fix', or 'update' may allow the type of modification to be classified as perfective, corrective or adaptive maintenance [Mockus00a]. Alternatively, identifying names within the string may be possible, thus providing more information than is obtainable through the author field alone. The ease and success of this will depend on the strictness of the comment field imposed by the project.

However, one significant piece of information is missing from this file, which is the initial length of the file. If an empty file was checked in at the start, then the current file size can be derived from the 'lines' attribute. However, if the first check-in was not an empty file, then this information can not be used. Checking out a file in order to measure the file size is the only means of accessing this information.

This only needs to be done once throughout the lifetime of the file however, and the results could then be stored within the repository for future reference.

4.6.2. Change logs

Although CVS is the most likely source of data, there are other relevant sources. Change logs are often provided when a new release of the software is made. Unlike CVS, there is no fixed syntax for the format of these logs. However, each project will normally retain the same formatting style throughout the log. Therefore, dedicated extractors could be written on a per-project basis to obtain information from this file.

Indeed, to do so would be extremely beneficial. The change log represents a human perspective of the changes made during the last version. These changes are usually given at a high level, representing changes to features and reporting fault fixes. However, for a developer, it will be more useful for these changes to also be reported at a lower level – for example, which files, methods and lines were modified in order to achieve the high level functionality. Parsing the change log, and cross referencing with comments in the CVS files may provide some benefit for completing this task.

4.6.3. Forums and mailing lists

Mailing lists present perhaps the richest source of information available to a developer. Each article may contain anything from a short fault report to a detailed explanation of the implementation of a specific feature. Alternatively, it may just be unsolicited e-mail. Generating complete information from these lists automatically is, with current technology, impossible. However, they could be used usefully if combined with other methods. For example, providing access within a tool to archives of the lists between given dates may provide more information of the changes occurring during a particular release. Similarly, searching for names within the archive matching the author of the patch may provide more detailed information about the changes that occurred.

4.6.4. Fault reports

Many open source projects will provide a dedicated fault forum, where bugs discovered can be reported. Often, these faults will come with a short explanation of the fault, and a subjective opinion of the severity of this fault. Importantly, these faults are usually numbered. Therefore, it may be possible to trace the progress of a particular bug through both mailing lists and CVS comments. A tool such as Bugzilla [BugZilla03], which is becoming commonplace with large open source projects, provides additional information. In this case, bugs are linked to other dependent bugs, showing, for example, how the existence of a bug in a file reader mechanism produces a further bug in the output. Bugs are also assigned to specific developers, possibly with an expected completion date. This information would allow a manager to see how many bugs are open or fixed at any time, and the load on a specific developer. Also, if fixed bugs are linked to the CVS comments, and so to a specific file, it may be possible to determine the reliability of each file by counting fix-on-fix rates, for example.

4.6.5. Summary

As can be seen, even without accessing the underlying source code, there are a large number of relevant data sources. Log files provide the names and dates and authors responsible for files that were modified, in a predictable format. Change logs provide high level comments as to the overall features and changed made within a particular release. Mailing lists may provide the reasoning behind a particular design decision. Finally, fault reports may possibly be linked directly to the log files, allowing the reason for a specific change to be identified.

The usefulness of each of these sources is increased significantly when cross-referenced with the others. Obviously, the feasibility of this will vary greatly on a project to project basis, depending on the processes employed within the project, and the ease by which this information may be extracted.

4.7. Summary

This chapter has introduced a new visualisation, Revision Towers, that provides a high level view of the evolution of software. A simple representation is used to visualise the contents of version control repositories, and therefore show how and when the files within those repositories have evolved. The type of maintenance that occurred may be hypothesised either through access to the comment fields within the repository, or by examining the relationship between header and implementation files. Finally, the evolution process is further emphasised through the use of animation.

The visualisation allows a number of questions to be answered about the project under investigation. A small number of the possibilities are described in more detail in chapter 7. However, the high-level view provided by Revision Towers means that many details of the evolution are not displayed. The following chapter introduces a new visualisation highlighting the changes occurring at the source code level instead.

ANIMATING THE EVOLUTION OF SOFTWARE

5. HfVis

5.1. Aim

The previous chapter introduced a visualisation that aimed to view the changes occurring within a project. This was done at a version level, allowing modifications to be seen at a high level. Although such a visualisation is useful for a general overview of a project, maintainers also require a more detailed view, with the ability to view the actual lines of code that were changed. Similarly, very few details were available about the structure of the files, and the relationships between them. This may make it difficult to determine the impact of a particular change.

The tool 'diff' is an integral part of most version control software and generates deltas to reduce space. As chapter two described, maintainers also use the tool in order to view changes between versions of a file, or between two separate documents. Within an open source environment, diff is also easily accessible, and often provides the only means of viewing modifications. However, as was also discussed, diff has a large number of shortcomings when used in this way. Visualising the output of diff presents a more comprehensible picture, although still has the same underlying problems.

The aim of this chapter is to introduce a new visualisation for viewing changes between versions of files. Unlike current tools, the output is no longer based on a line-by-line comparison. Instead, changes are considered at a syntactic level. This allows differences to be shown at a number of levels of granularity - between files, between classes, and between methods. Changes to the structure of the program may also be identified.

It is important to recognise that some of the benefits of diff will be lost. One of the strengths of diff is the ability to compare any two files containing text, with no other limitations. The new visualisation, HfVis, has stronger constraints. In order to allow more detailed analysis, the files to be visualised must be able to be parsed. This restricts the use of the visualisation to source code of a known language. In addition, the structure of the file must be syntactically correct, and so the visualisation may not be suitable for files in mid-development. The visualisation is also aimed at viewing changes in different versions of the same file, rather than two unrelated files.

As with Revision Towers, HfVis aims to show change across several versions of a file, rather than being limited to two or three as is the case with diff. In addition, the visualisation must be able to support an evolving data set. Some of the future proofing strategies set out in chapter three will be incorporated in order to achieve this.

The tool will ideally be placed alongside other tools provided within a configuration management system, so again it is necessary to restrict the visualisation to a standard display and interface.

As with the previous chapter, four aspects of the HfVis visualisation will now be presented. Firstly, the representation and layout algorithms used will be examined. This will be followed by looking at the role that animation plays within the visualisation. Finally, the available interaction mechanisms will be described. In addition, the forms of raw data that are available will be examined, to determine whether the approach is feasible.

5.2. Representation

HfVis is based upon a node and arc representation. However, a node contains much more information than a simple indication that a relationship exists with another node. Similarly, arcs also portray more information than the existence of a relationship.

The visualisation is aimed at visualising C and C++, as these languages are very commonly used within open source development. As C and C++ are not truly object oriented languages, there are many components that do not fit neatly into a class hierarchy. A change in a pre-processor statement or global variable in a single file may have a significant impact across the entire program, and therefore it is necessary to find a representation that provides these details. A class level view will lose these details, and so source code files are used as the initial grouping mechanism.

Although header and implementation files are designed to separate the interface from the implementation, in reality this is not always the case. Implementations may be brought into the header file for efficiency reasons, or to reduce build complexity. Therefore, a simple abstraction mechanism is used where header and implementation files are merged. A header file containing two separate class declarations, implemented in separate files, would be shown as a single node. This node will reflect changes made to both the header and the implementation files.

Such an abstraction mechanism is not always useful. In particular, a software library may have a very small number of header files acting as an external interface, and a large number of implementation files. This can be detected when the data is provided to the visualisation, and the abstraction disabled so each file is shown in a unique node.

An example node is shown as Figure 5-1. This will be used to demonstrate the main concepts of the visualisation. The component parts of this node (filename, file content, global metrics and the change circle) will be described in turn below.

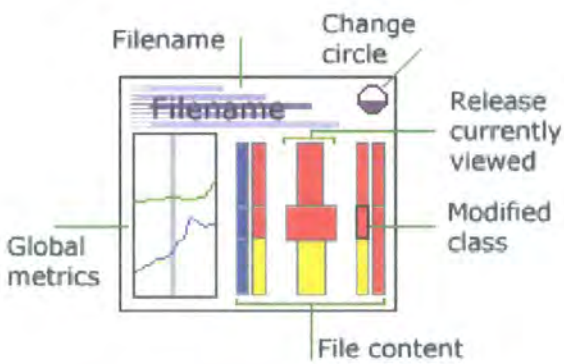


Figure 5-1. An example node from HfVis.

5.2.1. Filename

The filename is displayed at the top left corner of the node. The main part of the name is always displayed. The extension is displayed depending on the abstraction mechanism. If there are no

associated implementation files, or an associated header file, then the file extension is also displayed. This allows the user to interpret the contents of the node.

The filename is placed upon a background of five purple bars. The central bar is coloured darkest, and represents the current size of the file as lines of code, relative to the largest file in the project. The neighbouring bars represent the file sizes for the previous and next release of the same file, and are rendered in a lighter shade. The outlying bars are rendered in a lighter shade still, showing the releases before and after the previous bar. This allows the current size of the file to be seen in a wider context. Each bar also contains a white horizontal line. The length of this line indicates the number of existing lines within the file that were changed since the last release. Figure 5-2 shows the enlarged file size bars, with 1.0 as the release being viewed. The largest file size is reached at release 1.2, although release 1.1 has undergone the most change to existing code.

Figure 5-2. File size bars.

5.2.2. File content

The contents of a file are represented using a vertical bar, as shown on the right side of figure one. Five bars are again displayed, with the central bar representing the current release of the file. The bars to the right represent future evolution, and those to the left show historical information. As with the file size information shown with the file name, displaying five bars acts as a simple focus and context mechanism. The default behaviour is for the bars to the right to show the next, and next but one release, with similar behaviour for the bars to the left. However, the user may change this. For example, the bar at the extreme right may represent a specific release, and allow comparisons between the current release and the distant, future release. Alternatively, projects with very frequent release schedules may require a five-release gap between each bar, rather than one, to provide a better context mechanism.

A bar is partitioned into a number of vertical segments. Each segment represents a component within the file, and is coloured according to the type of component.

In C++ there is often little difference between identifying a component as a 'class' or a 'struct'. Structs may contain methods in the same way as a class. Therefore, the colours assigned reflect this. Red is always used to identify a class. A struct containing no methods is identified in yellow. A struct with methods is shown in orange, in order to distinguish it from a standard struct.

Global functions, variables and enumerated types are shown in blue, cyan and green respectively. Within a class, individual attributes are not visible. However, global variables may have a big impact on the execution of the software, and so they are given more prominence within the visualisation to reflect this. Global functions and types are also displayed for the same reason.

Additionally, #define statements are also displayed, in one of two ways. If the statement appears to be a constant declaration, the statement is coloured in purple. If, instead, the statement appears to be a macro declaration, then the statement is coloured in grey. The statement is categorised in this way to reflect the impact of any change. Changing a constant declaration to a different value may impact upon the behaviour of the program, but the pre-processed source code will remain almost identical. However, changing a macro declaration may have a large impact on both the behaviour and the pre-processed source code.

Finally, typedef declarations are shown in pink, and union declarations are shown in brown. For convenience, the colours used are summarised in Table 5-1.

Section colour	Structure represented
Red	Class
Orange	Class/Struct hybrid
Yellow	Struct
Green	Enumerated type
Cyan	Global variable
Blue	Global function
Purple	#define as a constant
Pink	Typedef
Brown	Union
Grey	#define as a macro

Table 5-1. Colours used to represent structures.

Various orderings within the bar are possible. The default ordering is by component type, with the elements sorted alphabetically within this. This highlights changes to the name or the type of an element. For example, if a component changes from a struct to a class, the component will disappear from the struct section, and reappear in the class. Alternatively, a plain alphabetical ordering is possible, which is much more sensitive to name changes. Finally, a creation ordering is possible, where new elements will always appear at the bottom of the bar. This is the most consistent view, providing that there are a minimal number of deleted elements.

```
#ifdef HEADER_H
#define HEADER_H

#include <pair>
#define MAXSIZE 255

struct Coord
{
int x;
int y;
};

typedef pair<Coord,Coord>
CoordPair

class C1
{
protected:
CoordPair method1();
public:
Coord method2(CoordPair p);
bool method3(CoordPair p);
}

#endif
```

Bar showing class C1, struct Coord, typedef Coordpair, and #define MAXSIZE respectively.

Note the ordering does not reflect the order within the file. The contents of the components are not shown at this level of detail.

Figure 5-3. A short header file, and the resultant file content bar.

5.2.2.1. Component size

The height of each segment is a weighted percentage of the size of the segment related to the size of the file. Size is considered in one of two ways, and may be controlled by the user of the visualisation. If abstraction is used to group header and implementation files, then the size of the composite file is used.

Size can be considered as the length in lines of code of the component, and the lines of code for the full file. The default behaviour is that global variables, enumerated types and #defines are excluded from these calculations, and displayed as a fixed size. Simple structs are allocated one line of code per attribute, regardless of the actual formatting. No additional behaviour is applied for global functions, hybrid structs, or classes.

Alternatively, the size of a component can be considered in terms of the number of methods and attributes. In this case, global functions, variables, typedefs and #defines are allocated a size of 1. Enumerated types, unions, structs and classes may be considered as normal. However, the user is given further control of this in order to modify the importance of a particular component. For example, it may be that the user is less interested in private methods, and so allocates a greater weight to public methods within the class.

5.2.2.2. Component metric

Of the five bars shown, the four non-central bars are shown with fixed width, with the components within those bars also of fixed width. The central bar, representing the current version, is shown significantly wider than the other bars. In addition, the width of each component within this bar is also variable. This allows an additional metric to be displayed.

For example, if the height of the component represented the number of methods, then the width of the component could represent the size in lines of code of the implementation of those methods. A change in the width over time would indicate that code had been added to or deleted from the component.

Alternatively, the width may be used to indicate the total number of changes that occurred within the component during the last release. Unstable components could then be recognised, as the width of the component would vary wildly over time. Stable components would have a narrow width. A further option is to calculate the number of changes over several releases, and display the average. This provides a smoothing effect, and may be more useful for components undergoing infrequent change.

There is no reason for the width of the bar to be directly proportional to the size of the metric. Instead, the user may control the exact mapping. For example, it may be that the user is more interested in whether any change occurred than in the extent of these changes. In this case, there should be a significant visual difference between no changes and a single change, with subsequent changes shown as smaller increases.

5.2.3. Global metrics

The vertical bars are used as a small focus and context mechanism. The central bar provides the most details about the current release, but the neighbouring bars also show some details. These details are still centralised around the current release however.

In order to provide a better overview of the entire project, a line graph is used. This is displayed by default at the left side of the node, as shown in Figure 5-4. The x-axis is used to represent releases, and covers the duration of the project. The release that is being focussed upon within the node is shown with a vertical grey bar. The y-axis plots the value of a metric at that point. The graph can then be used in co-operation with the file content bars to display two different forms of data simultaneously.

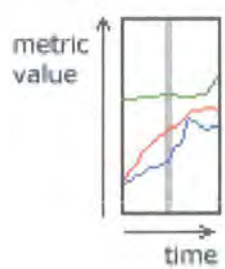


Figure 5-4. Global metrics graph.

Local data: Metrics showing details of the current file may be shown. For example, plotting the current file size relative to the maximum file size would highlight growth. Similar measures could include the number of methods or constructors, or the average complexity within the file. Alternatively, metrics may be drawn from the configuration management system. This would allow details such as the total number of authors contributing to the file, or the number of versions within a release to be shown.

These details may be plotted relative to the file, or relative to the largest file within the project. The latter case is more suitable for highlighting extreme results.

Project data: Relevant project data may again be derived from the source code or the configuration management system. Source code metrics may show the total number of files in existence or a measure of coupling within those files. Configuration management metrics may show the number of fixed and open bugs within the software, or the extent to which the software is used.

The benefit of integrating the graph into a node is that graphs may be localised to different nodes. Rather than presenting a summary of the entire project, the graphs allow different areas of the project to be considered separately. The file content bars then provide the opportunity for further exploration of any issues raised by the graphs.

5.2.4. Recent changes

Within software development, a tiny change may have a huge impact upon the entire program. Therefore, it is crucial that small changes are not hidden due to the abstraction techniques that are applied. This is handled using two methods within the visualisation.

Firstly, any change that occurs within a component is emphasised within the file content bars by giving the component a thick black border, as shown in Figure 5-5. As the number of changed components within the project is likely to be a small percentage of the entire size, the border ensures that those changed components are obvious to the user.

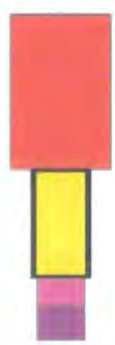


Figure 5-5. File content bar, with change identified within the struct (yellow).

The second method indicates the last point at which a file represented by a node was changed. This ensures that the user is also aware of recent, as well as current, changes. A small circle is displayed within each node, which is filled to represent the most recent change, as shown in Figure 5-6.

Figure 5-6a shows the state of the circle for a node that is currently undergoing change. Figure 5-6b shows that change occurred in the previous release, 5.6c and 5.6d show that change occurred in the releases prior to that. Finally, 5.6e shows that the node was not changed recently.

Figure 5-6. The change circle.

The combination of these techniques therefore gives most emphasis to recent modifications, to allow these to be further identified by the user. However, by retaining a small degree of change history, it reduces the likelihood that a developer will miss a change.

5.2.5. Program structure

The content of the node has already been discussed. However, unlike Revision Towers, information about the structure of the program is also available from the data source. Using this allows some semantic, rather than just syntactic, differences to be viewed and represents a significant improvement over lexical file differencing tools.

Within HfVis, two structural sources are available directly. A C or C++ file uses pre-processor #include statements to indicate the source of methods and classes used. These statements will usually refer directly to other header files. As the header file is an important part of the HfVis visualisation and is used by the abstraction mechanism, this provides a computationally cheap means of identifying a basic call structure within the program. The second alternative is to extract a static call graph using the fact extractor. This may be more useful than just using header files, but will lead to a more complex graph as a result. Other abstractions based on a call graph, such as a dominance tree, could also be used. However, these have not been considered in HfVis as the prospective users are unlikely to be familiar with such an abstraction.

The main difficulty of extracting structural information is that pre-processor statements may have a significant impact upon the call graph of the program. In particular, many open source projects are ported to several platforms and compilers. These are almost always controlled by #ifdefine statements, for which suitable values are provided by a make file or a compiler. In many cases, cross platform development will use functions with the same name, parameters and general behaviour, but these will be contained within different files. This provides further difficulties when using a call graph, as although the function called is easily recognised, it may be difficult to determine exactly which file contains that function.

A possible solution to this is to force the user to declare the necessary #define values before using the visualisation. This then allows the fact extractor to build up an accurate picture of the files actually used, and so it is clear where a particular function is contained. However, there are two problems with this approach.

Firstly, the underlying aim of the visualisation is the same as every other file comparison tool – to show all of the changes between two files. This solution changes this aim, meaning that it will be necessary to provide every possible combination of #define values in order to guarantee being able to view every change that occurred.

The second issue is that the #ifdefine conditional statements may change over time with the rest of the code. Therefore, there is no guarantee that providing a particular value will result in the desired behaviour throughout the duration of the project. For example, in early versions a large number of #define statements may exist, with no fixed naming scheme. After later refinement, prefixes may be added in order to reduce conflicts with other code. Therefore, the user of the visualisation would have to provide values for the statements on a release by release basis, which is error prone and time consuming.

An alternative solution, and one employed by HfVis, is to essentially ignore the pre-processor statements. Instead, two different representations are used, depending on the potential influence of the pre-processor statements. If a statement block is surrounded by a #ifdefine, #endif condition, the block is assumed to be affected by the pre-processor, and so represented differently from a block that is not surrounded. The exception to this rule is the main guard that is almost guaranteed to appear at the top of a header file that prevents the contents of the header file being included multiple times.

The representation uses coloured lines to visualise the #include information. As with the rest of the representation, historical information is also included in order to provide context. Two files linked by a #include will be shown with a two colour line – cyan at the source, and green at the destination. Two colours are used in order to indicate the direction of the line, and so the need for arrowheads is eliminated. If the #include relationship was new for this release, then the line will be shown brightly, to highlight the new change. If the relationship is older, the line will be faded, depending on the age of the relationship. Fading also has the additional effect of reducing the clutter in the graph, as older relationships will have a lower emphasis than other, more important, objects.

However, consequently it may not be apparent to the user if an old relationship is removed, as the effect would be to remove an already faded line. Therefore, in order to give emphasis to this, deleted #include lines are also shown. In this case, a two-colour line is again drawn, from orange to red. If the deletion occurred in the current release, the line will be shown brightly. This will then fade over the next two releases, and will then no longer be shown.

Relationships that are unaffected by the pre-processor are shown as a solid line. If the pre-processor may be involved in the include process, a broken line is displayed instead. The pre-processor statements affecting the include process may be shown on request. Figure 5-7 shows the various possibilities that exist. File1 has recently included file2, and stopped including file4. Additionally, file3

has been included for a long time, as shown by the faint colour, although the inclusion is dependent on pre-processor statements, as shown by the broken line. Finally, file1 also includes two external header files, and has stopped including one a few releases ago, as shown by the short lines at the top left of the node.

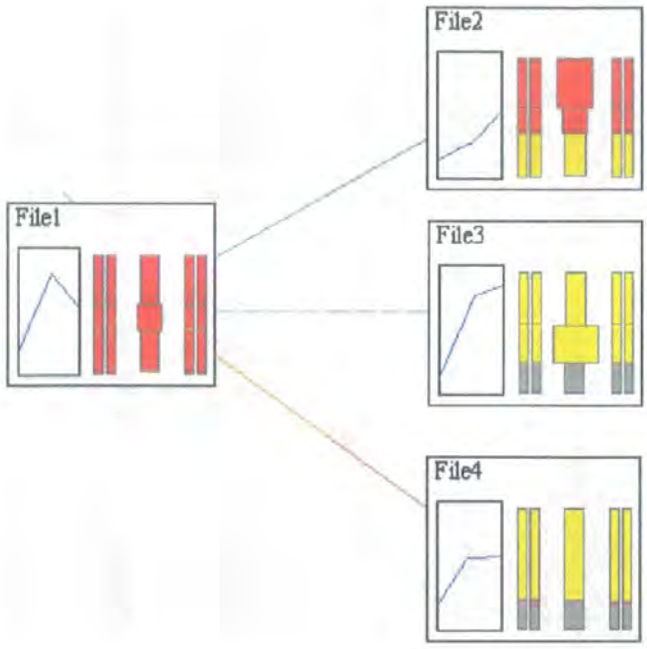


Figure 5-7. #Include graph, showing the relationship between four files over time.

5.3. Layout

Chapter 3 discussed the importance of the layout algorithm when working with evolving data sets. HfVis is no exception to this. As with Revision Towers, a future proof strategy is required, in order to present a consistent picture to the developer each time the visualisation is used.

The layout is based on a node and arc graph. Although there are a number of well discussed problems with such a representation, it is an easily understood technique for displaying any sort of relationship diagram. Initially, a force directed layout is used, as this has reasonable support for familiarity. In order to be consistent, the layout will always use the first release of the software as a basis, and no future knowledge will be used. The layout will then evolve predictably as new files are introduced. Such an approach is feasible provided that large structural changes do not occur over a short period of time. Where this is the case, a large layout change may occur which will be difficult to track. In order to reduce the impact of large layout changes, it is possible for the user to lock selected nodes to ensure that their location will not change. It must be recognised however that this may create a less optimal graph. However, realistically, this situation is unlikely to arise often in the duration of the project. Therefore, although the worst case for the layout is very undesirable, the choice of layout was weighted towards a beneficial layout for the majority of the time.

A radial layout is also provided as an alternative to the force directed placement. Again, this layout algorithm is used because of the support for evolution. The purpose of this layout is to allow close investigation of the structure of a specific file selected by the user.

The radial layout is emphasised using a number of concentric circles. The file under consideration is placed at the centre. Files that it includes are placed on the next ring. The process is repeated for the new files, so that the include structure of a depth of 2 is generated for the selected file. All other files not included are placed on distant circles, with no relationships shown between them.

The user may then expand the graph by expanding specific nodes, in a similar fashion to a tree view. Repeated nodes are handled in two separate ways. The current node being expanded may link directly to the existing node. This may reduce complexity, although the graph becomes more difficult to read as assumptions about relationships based on the location relative to the current node no longer apply. The alternative is that the same node may appear more than once in the graph. The entire node is duplicated to allow direct access to the information contained within it. However, the node is also faded to highlight the fact that it is a duplicate node, and so reduce the emphasis given to it.

Figure 5-8 shows an example graph, generated to a depth of 3. The starred nodes (added) indicate the faded, duplicated nodes. In this case, the extreme left and right nodes can be seen to be a duplicate of the top node. The small box at the bottom of these nodes allows quick access to the original node.

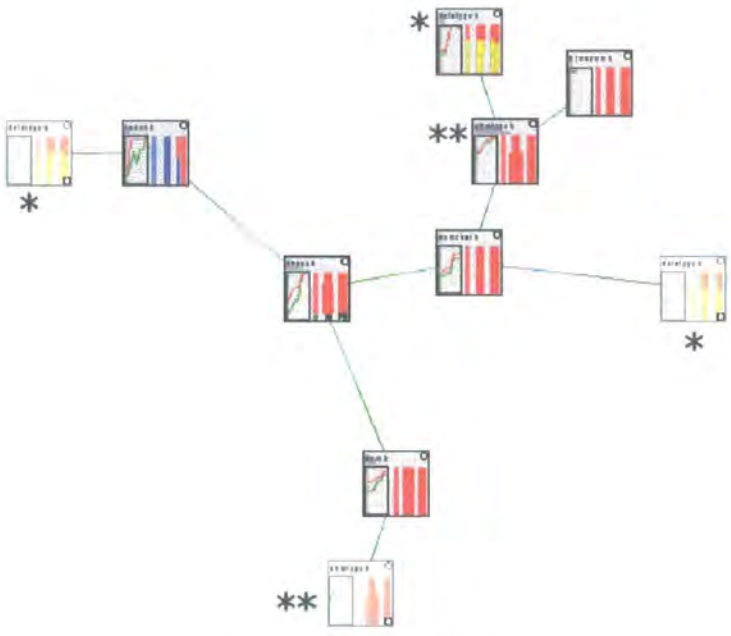


Figure 5-8. Example radial graph layout, showing duplicated nodes.

5.4. Animation

Animation adds an important dimension to the visualisation. Rather than just using animation as an information hiding mechanism, or for generating smooth transitions between static pictures, HfVis uses animation to display additional information that is not otherwise apparent.

The description of the representation and layout was based on visualising the file contents based on a single release of the software. However, as with Revision Towers, it is also possible to view the contents of the file changing over time, with the nodes within the visualisation changing in order to reflect this. The primary motivation of adding animation to HfVis is that movement should be used to indicate that the moving object has undergone an underlying modification in the software. The human eye is aware of movement and change even without giving full attention to the area under consideration, although the change must be relatively sudden to avoid change blindness issues. Therefore, by associating movement or change in the visualisation with change in the software, the user is made directly aware of the modification.

Unfortunately, it is very limiting to use movement to only convey that the moving object has changed in some way. Such an approach means that no resizing or repositioning may occur. This is because in order to reposition a changed object such as a new node, it is likely that other unchanged objects must move in some way to accommodate this. In order to resolve this issue, some overloading is necessary. This overloading is managed within HfVis by separating the animation into three phases.

The first phase uses movement solely for layout changes, such as creating space for a new node that will be introduced to the view during the release. The change circle and the arcs within the graph should also be updated during this stage. The last phase is also restricted to uninformative changes, where space that was created by a node being deleted may be closed. The central phase represents the actual modifications that occurred between the previous and current release. Here, any movement that occurs is used only to show that the underlying software has been modified in some way.

The animation created is bi-directional. As with Revision Towers, this requires more than just pre-rendering a sequence of frames that may then be viewed in any order. For example, the user may view an include graph in the current release. It must then be possible to look at the same graph based on previous releases, in order to determine how the graph came to be this way. If nodes within the graph are then expanded, the facility must be provided to view that same expansion in earlier and later releases. This would not be achievable if the sequence were pre-rendered.

In order to support the expansion functionality, it must be recognised that the structure of the graph may change over the time specified. Therefore, the term 'same expansion' must be clarified. In HfVis, if the structure of the graph changes then nodes are expanded in the new graph if they were expanded in the old graph and there is also a direct path between that node and the root in the new graph. If it is necessary to expand other nodes in order to include an expanded node, this may be done provided no more than two nodes deep need to be expanded in this way.

As with Revision Towers, a timeline is used to allow the user to observe the current position in time, and to pause or play the animation. An example timeline is shown in Figure 5-9, and uses the same colour scheme as Revision Towers. As can be seen, the timeline is made up of two parts. The top part spaces the releases according to the release date. The bottom part spaces the releases equidistantly. The black line then indicates the current release within the visualisation.

During the animation, each release is allocated the same number of frames, and has the same frame rate. The black line will therefore move at constant speed in the bottom part, and with variable speed in the top part.

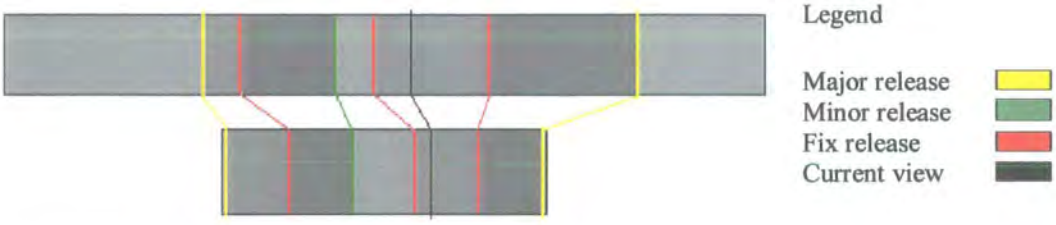


Figure 5-9. HfVis timeline, showing six releases.

5.4.1. File size

As previously discussed, the current file size is represented using five purple bars behind the file name. The central bar always shows the 'current' release. Therefore, as the evolution of the project is viewed, the 'current' release will change. In order to represent this, it is necessary to update the values of the bars after each release is viewed, as they will refer to a different release.

There are two alternative means of achieving this. The first is that the bars scroll vertically behind the filename, as shown in Figure 5-10.

Figure 5-10. Showing filesize change with scrolling.

Although this may be a suitable representation in general, it has been rejected within HfVis. The main reason is that the same movement will occur regardless of whether any change occurred in the releases concerned. Instead, the method shown in Figure 5-11 is used.

Figure 5-11. Showing filesize change with growing and shrinking.

The benefit of this approach is that the movement is now tied to underlying change. If the file in the current release has grown, then the bar will extend to show this. If the file has remained the same size, then the bar will not change.

In addition, the same number of frames of animation is allocated to the extension process for each node. This is then split into two stages, to show the number of added and deleted lines separately. The larger of these two values is shown in the second stage, to enable the user to determine whether the overall file size is larger or smaller. As a fixed number of frames are used, the speed at which the bar extends will highlight the growth of the file. If the bar extends rapidly, then the file size has grown considerably. Linear interpolation is therefore used rather than a slow in-slow out interpolation, to avoid high acceleration being perceived as significant growth.

5.4.2. File content

File content is represented using the five vertical bars at the right side of the node. This representation is also animated within HfVis. As with the file size, the central bar represents the current release, which will need to be updated as the visualisation is viewed through time. Scrolling the bars horizontally was rejected as a suitable mechanism for the same reasons as above. Instead, each bar is updated individually to reflect the changes made for that release. Three types of change may be viewed.

New and deleted components are shown by adding or removing a segment from the bar. As the bar represents the makeup of the file as a percentage, adding a component will, by necessity, resize the other segments to create space for it. Similarly, space will close up over a component that is deleted. The size of the new or removed component will impact upon the degree of movement. If a large class were removed from the file as part of a restructuring operation, the bar would change significantly to show this. Removing a small function will result in a very small change to the bar. It can be seen from this that the aim of movement representing underlying change is only applicable to the bar as a whole, rather than the component parts.

Resized components are shown by increasing or decreasing the space provided to them. However, it must be recognised that this is always shown relative to the file as a whole. Therefore, if every component within the file increases in size by exactly the same amount, then no changes to the height will be seen. However, the dark border of the segment will still indicate that some change has occurred.

Finally, change in the make up of the component may also be viewed. For a struct, class or enumerated type, this reflects a change in the number of methods or attributes. For global functions, this reflects a change in the number of parameters. For other types, it reflects a change in the value. The modification

is shown by lightening the segment if a new method, attribute or parameter is added, and darkening the segment if one has been deleted. In the case of a changed value, the segment is lightened.

The darkening and lightening process is split into two sections of equal time to show new and deleted aspects separately. Within each section, the effects are cumulative. Therefore, a class with five new methods will be shown lighter than a class with just one. The lightness and darkness mappings are inversely exponential, so that the greatest difference in lightness is seen between a class with the same methods, and one with a new or deleted method. A linear interpolation method is again used to generate smooth transitions.

Figure 5-12 shows a complete animated sequence for a release over five frames, with three file content bars. A chronological layout has been used. The sequence shows a file with a #define (purple), two global variables (cyan) and four global functions (blue) changing to one where the functions have been replaced by a class (red). In addition, the second variable (starred) has changed value, as it is lighter, and the #define has changed in some other way as indicated by the black border.

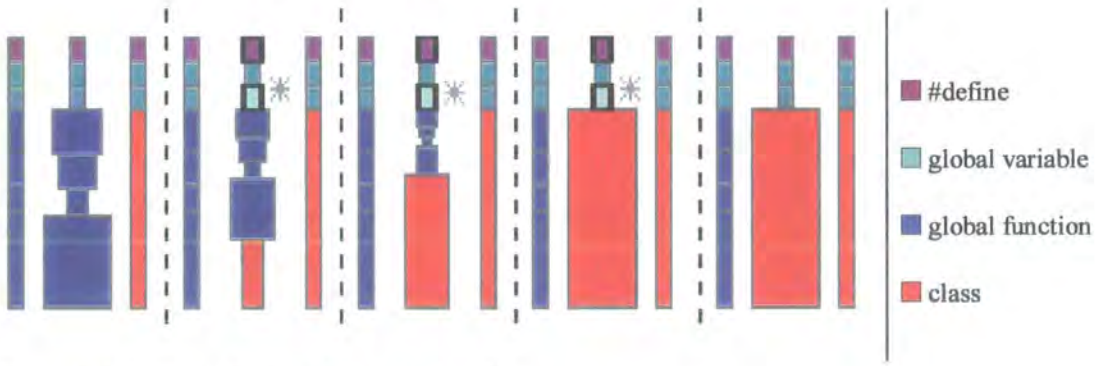


Figure 5-12. Example of animated sequence for a file.

5.4.3. Include graphs

As stated earlier, the include graph may be laid out using a radial or a force-directed algorithm. Animation is used to update the layout to reflect new and deleted files, and new relationships between those files.

New nodes are introduced to the visualisation over a period of time. Firstly, space is created for the node within the first animation phase. Once the space exists, a dark green border is then displayed, and the node is then gradually faded into this space over a period of two releases, during the second animation phase. The green border remains during this fading in process. Similarly, deleted nodes are shown displayed with a red border, and faded out gradually over a period of two releases. The layout algorithm will then reclaim the space during the final animation phase. The purpose of the borders is to highlight the areas of change, and reduce the dangers of change blindness.

This sequence is demonstrated in Figure 5-13. Here, four frames of animation are shown. The first frame represents the system before any animation occurs. The file represented by the node with the red border no longer exists, and the node is about to be deleted. The second frame shows the result just after the first animation phase, with nodes moved to create space for the new node, indicated by the

green border. The third frame shows the result at the end of the second phase. The new node has been partly faded in, although the file does not exist for another two releases, as shown by the lack of a central file content bar. The old node has been faded out, with just the red border remaining. Additionally, the file content bars have been updated as normal, as shown in the top left node. Finally, the fourth frame shows the state at the end of the third animation phase. The red border has been removed, and the layout has been updated.

For the force directed algorithm, the nodes are relocated using a simple linear interpolation between the old and the new layouts generated by the algorithm. For the radial algorithm, the interpolation technique depends on the depth of the tree created, configurable by the user. For shallow trees, a linear interpolation is again used. However for deep trees, a polar interpolation is used instead where a node remaining at the same distance from the root will move to a new location by traversing the current ring. If the circumference of the ring is too small to incorporate all of the nodes, the radius of the ring may also gradually increase to allow more nodes to be added without overlap.

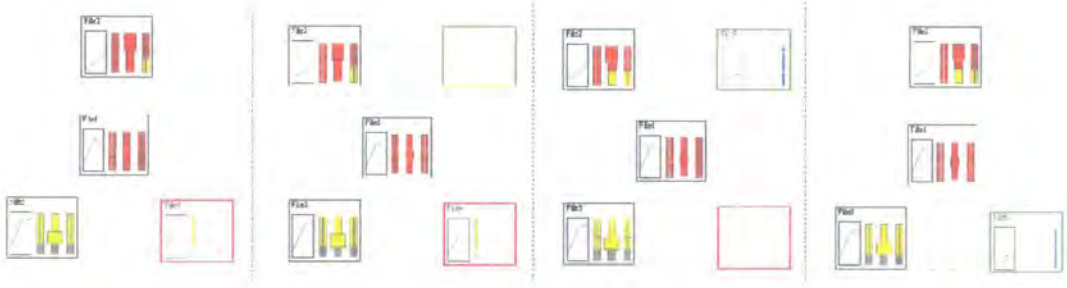


Figure 5-13. Example of files being added and removed.

5.4.4. Issues

Although the animation is an important part of the visualisation, there will be occasions where the animation will not be viewed. In particular, one of the roles of visualisation is to provide a basis for communication. Within a shared environment, this communication may occur within online forums with poor support for interactive animated images. Therefore, a snapshot of the animation must be as informative as possible, with minimal misinformation.

The historical and future bars play an important role in increasing the information content of the static picture. Although the changes are more obvious and informative when animated, the history bars for both file size and file content allow some form of comparison to be made between the current release and a previous release statically. Without the history bars, the static image of the visualisation would be almost meaningless as a means of identifying change.

However, it is still possible to misinterpret a static image. For example, consider the following situation, where version 1 of a file contains a class, and struct A, and version 2 contains class, and

struct B. Both structs are the same size. The basic animation sequence would then appear as shown in Figure 5-14.

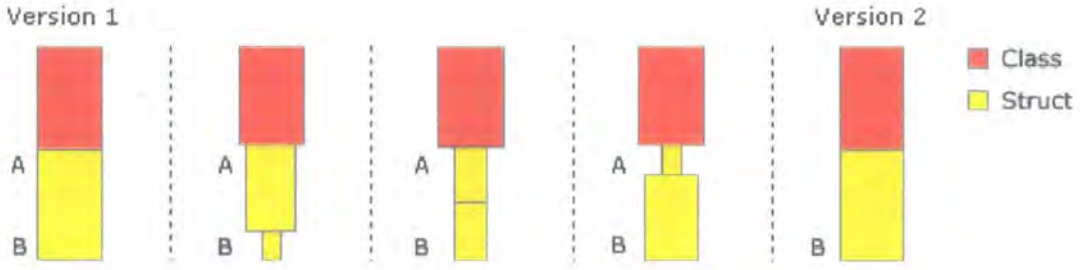


Figure 5-14. Example of sequence with identical start and end frames.

However, if only the start and end frames are examined, it appears that there has been no change. In this case, the change circle displayed on each node acts to ensure that the fact that a change occurred is not lost. Although it is not possible from this to determine exactly what has changed, the circle indicates that the two releases are not identical. This allows the user to investigate the file in more detail.

5.5. Interaction

HfVis supports a number of interaction techniques to allow the user to explore the data. The user may interact with the visualisation at any point, with the animation updating seamlessly to reflect any results of the actions.

5.5.1. Zoom

The size of the node used within HfVis means that it will not be possible to view all of the nodes simultaneously in full detail. Therefore, a smooth zoom operation is available to allow the user to focus on a smaller number of nodes. This is particularly useful with the layout algorithms available, as both have the effect of clustering nodes together. Therefore, the zooming operation allows the user to concentrate on several connected nodes in more detail.

Providing level of detail functionality enhances the zoom operation. The level of detail concept is common within 3D environments, where a low detail texture is used for distant objects and replaced by a high detail texture when the object is closer to the camera. A similar idea is used for the zoom operation within HfVis, as demonstrated in Figure 5-15.

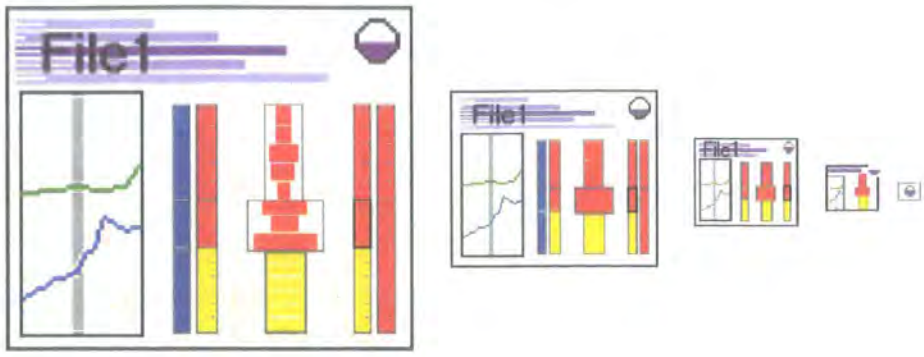


Figure 5-15. Five different levels of detail for a node.

The level of detail behaviour has two benefits. The first is that it provides a more useful image for the user when they zoom out. By reducing the data displayed within the node, the remaining content may be allocated a greater percentage of the space. If the same content were used when zooming out, it would become increasingly difficult to read any details, as the objects become too small. Using the level of detail technique therefore means that less important details are lost entirely to ensure that the important details remain legible.

The second benefit is that it reduces the complexity of the display. As the zoom level decreases, the number of objects shown on screen will increase. Therefore, the number of moving objects on the screen will also increase. In order to prevent the display from being animated to the extent that it is not possible to determine any meaningful information, it is necessary to reduce the movement that occurs. This is achieved by removing extra details that are not crucial to the understanding of the visualisation, such as the historical information. An additional benefit of this is that the computational complexity is also reduced when larger numbers of nodes are shown. It is important that the frame rate within the visualisation remains constant, as each release is allocated the same time period. Zooming out will generate a more complex image, as more data will be visible, and so will also lower the frame rate. A decreased frame rate may create the impression that the time period of the release was extended in some way. In order to avoid setting the frame rate to this lowest denominator regardless of what is displayed on screen, providing a level of detail setting allows the frame rate to be set to a higher, smoother, level.

The primary difference between this and a traditional level of detail implementation is that the change in detail is much less subtle. There will be a small jump in content as objects change from one detail level to the other. This could be resolved using animation, in order to resize or fade out unnecessary items. However, zooming in the second animation phase would then use animation for both information content and layout changes, creating confusion. Instead, although zooming can take place at any point, level of detail changes are not made until the central animation phase has been completed.

In order to maintain context when zooming, a small overview window is also provided. The layout of nodes within the main view is replicated within the window, and nodes are shown at the smallest zoom level. Any change that occurs within the node is also reinforced by giving that node a heavy border.

5.5.2. Selection

Within HfVis, nodes are displayed against an off-white background. If a node is selected, the node is shaded in a faint colour, chosen by the user. This allows the user to highlight several categories of interesting nodes as part of ongoing investigations and acts as a simple classification and annotation mechanism.

Nodes may also be selected as a result of executing queries. For example, the user may wish to highlight all files that have been edited by a particular author at any point in the lifetime of the file, or that have been edited by a particular number of authors. Again, the user may select a colour to be used for the result of the query.

Further information may be discovered from nodes using a mouse over operation. Every part of the node provides further information on demand. A mouse over on the global metric display, or on the file size bars, will show the exact value of the point that has been plotted, together with the highest and lowest values. A mouse over on the name will display further details about the node, such as the files used to generate the information within that node, the current release number, and authors involved within the file.

A mouse over on a segment within the file content bars will show further details of the changes that occurred within the release, including the name and size of any changed components. Figure 5-16 demonstrates tool tips containing summary and detailed information that can be displayed for a class segment, depending on user preference. Red and green indicate deleted and added elements respectively. Yellow indicates that the type or value of the element has been changed. Finally, the height of the bars in the summary tool tip indicates the method size, and a curved edge indicates the method has been changed. Items are ordered chronologically within the summary view, and retain the same position.

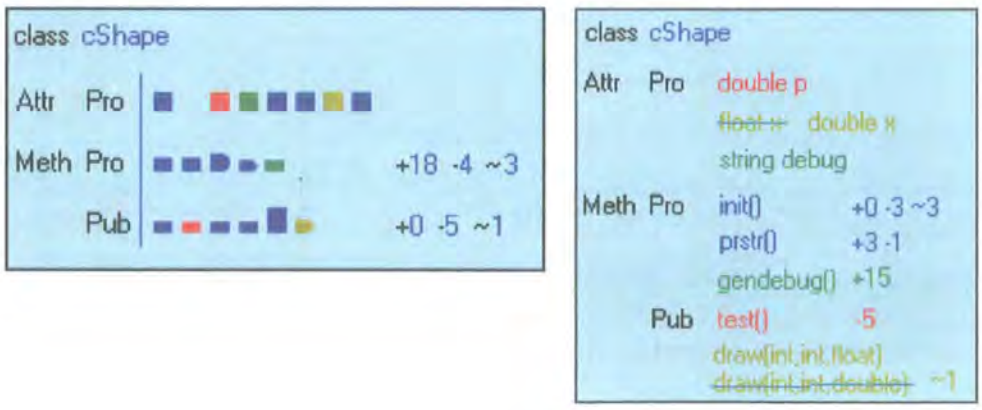


Figure 5-16. Tool tips displayed with mouse-over on a class.

The same technique may be used to request a standard diff-based view of the changes, which is shown as a further window. This view then allows access to the source code in order for the user to be able to see the exact changes that were made. This view may be locked to a specific release, or may change as the animation progresses.

Finally, a mouse over on a link between two nodes will highlight that link by bringing it to the foreground and re-colouring the link to make it brighter. This is particularly useful for old links that are usually shown as faint lines. In addition, the #defines that determine the conditions associated with the relationship are also listed when applicable.

5.5.3. Filtering and grouping

HfVis has a fluid layout that is well suited to filtering and grouping operations. Nodes may be filtered at two levels, as a result of previous selection operations. In the first case, the size of the node is reduced to the name and change circle only. Links to and from the node are still shown, and the node may still be involved in further selections. At the second level, the node is removed from the visualisation completely. Links to and from the node are reduced to very short lines at the source node, indicating that a link still exists.

Grouping operations involve grouping according to predetermined characteristics, such as alphabetically, or by using the results of the previous selection operation. A grouping operation may only take place during the pre-animation stage. Grouped nodes are laid out in a new window using a spiral layout. In order to maintain context, nodes are copied sequentially from their current location to the new location on the spiral using animation. Any nodes that are not part of the group are placed on a distant ring around the spiral. Links between nodes are again reduced to short lines, as with the filtering operation. The user may continue examining the grouped arrangement over time, and nodes may join or leave the group as before. If nodes within the group are reordered – for example, if nodes were ordered within the spiral according to the number of changes made - then a node will move to the new position by traversing the spiral.

5.6. Code-level View

An alternative view available within HfVis is a code level view. This allows the changes that occur to be viewed at a lower, source code based, level.

A SeeSoft pixel view is used, where each line in the class is visualised by a coloured pixel. The colour allocated to a pixel is chosen to match the colour used within the file content bars in the main view. For example, a line of code that belongs to a global function in the file level view will be displayed as a green pixel. The exception to this is pre-processor statements that are not shown in the file content bars, such as #includes. These are shown in grey.

Pixels are left empty between two classes, functions or methods in order to group pixels appropriately. The intensity of a pixel determines the age of the line, with a recent change resulting in a bright pixel. This pixel will then fade exponentially over time if no further changes take place. The purpose of this is to allow recent changes to be spotted easily by the user. An example of a node showing the file described in Figure 5-3 is shown below as Figure 5-17. The block above the dotted line shows the header file, and the block below shows the associated implementation file, or files.

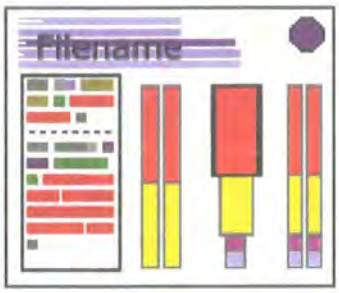


Figure 5-17. Pixel view of file contents.

The pixel view is animated to allow changes over several releases to be seen. The animation is again split into three phases. Within the first phase, space is created for any new code that will be incorporated. This new code is then faded in during the second phase. Any code that has changed is repainted with a bright pixel. Code that has been deleted also remains temporarily, but is coloured in black. The deleted pixels are then removed during the final phase. This therefore continues to reinforce the concept of animation being used to indicate underlying change. A mouse-over operation on a pixel will show the actual line of code, and a series of releases and authors showing when it was changed and by whom. In the case of changed code, the change that occurred will also be shown.

One change that may occur within the source code is that code may be moved or copied from one file to another. The representation will show this differently from entirely new code. Instead, the new code will appear to come from the corner of the view, and then expand in the space provided. Similarly, any code that was cut will be contracted, and move to the corner of the view. A mouse-over operation on the selected code will reveal the source, destination and content of the code. Clones of five lines or more will be considered in this way.

The view is contained within the main node. The user may choose to replace the file content bars or the global metric display with the pixel view either globally, or on a node by node basis. The view also supports varying levels of detail dependent on the zoom level. At low zoom levels, pixels are clustered so that a single pixel may represent several lines of code, although these lines must all make up the same function. The brightness of the pixel is determined by the most recent change occurring within any of these lines. At high zoom levels, the pixel view may be replaced by a line by line view. As switching between the pixel and line based views automatically may be disorientating, the switch may be controlled by the user.

5.7. Class-level View

The representation defined above has been designed for viewing changes within the program at the level of files. However, it is also possible to use a similar representation for use with classes. This makes the approach suitable for languages such as Java, where object oriented development is enforced.

The node remains the same as before, although the edges of the node are rounded. This acts to distinguish the two types of representation. The file content bar is also separated differently. The bar is split into six sections, and coloured according to Table 5-2:

Section colour	Structure represented
Purple	Private attribute
Blue	Protected attribute
Cyan	Public attribute
Red	Private method
Yellow	Protected method
Green	Public method

Table 5-2. Colours for a class-based representation.

The size of each section is again calculated as a percentage of the whole class. Size is considered either as lines of code, or the number of methods or attributes in each class. The size may be weighted, to give greater consideration to public methods and attributes, for example.

A class level view also provides the option for a further graph layout. In HfVis, the inheritance structure of the class may be mapped onto the radial graph layout, in a similar way to the include graph at the file level. Nodes may again be expanded to allow more of the class hierarchy to be viewed. The concentric circles within the radial layout are also shown in a different colour to distinguish between the inheritance graph and the include graph.

5.8. Available Data

The visualisation will use the data provided by a source code fact extractor as the primary data source. Facts will be generated from each release of the software, rather than each version of a file. The reason for this is twofold. Firstly, the amount of data required, and so the computational expense, is significantly reduced. Secondly, during the development before the release, it may be unclear which versions of the files should be selected in order to build the program. Therefore, it will not be possible to determine with any accuracy the structure of the system. When the software is released, the versions of each file used are known, and so the structural information may be derived. However, the effect of

this is that some interesting version information, such as the developer involved in a change, is unavailable to the visualisation.

The amount of data available to the visualisation will depend greatly upon the fact extractor used. However, for the visualisation to be useful, the following should be provided:

File level: The extractor should provide the names of the files within the project. In addition, it should also provide details of the components implemented within that file, such as class or function names with parameters, global types and variables, and pre-processor statements. Some means of access to the underlying source code is also necessary.

Class level: The extractor should provide the classes defined within the project, and related inheritance information. The methods within each class should also be available.

Method level: The extractor should provide basic details about the method, such as the parameters, access, and size. Call graph information should also be available.

Any metrics generated by the fact extractor during this process may also be used by the visualisation. In addition, data from the configuration management system may also be used, if provided. Although specific version information is not required, release level data is valuable. For example, the time of release, the number of versions making up the release, and the authors involved within the release are all relevant.

The visualisation is targeted for use with C and C++ code. However, other imperative languages, such as Java, would also be appropriate at various levels of detail.

5.9. Summary

This chapter has introduced a new visualisation, HfVis, which allows software modifications to be viewed in the context of the project as a whole. Specifically, changes are identified within software structures, as opposed to only providing the line number of a change. An augmented node and arcs representation is then used to visualise the modifications, and allows the effect of these to be shown over time. Finally, animation is used to allow a series of releases to be viewed in detail, whilst minimising the impact on the layout and representation used.

The visualisation also allows a number of questions to be answered about the project under investigation. A small number of the these possibilities are described in more detail in chapter 7. In particular, the low-level view provided by HfVis allows further investigation of areas of interest identified when using Revision Towers.

ANIMATING THE EVOLUTION OF SOFTWARE

6. Implementation

6.1. Introduction

This chapter provides an overview of two proof of concept tools, demonstrating the feasibility of the visualisations presented in the previous two chapters. The animation system used to power the tools is introduced, and some of the limitations that affect the later implementation are described.

The implementations of Revision Towers and HfVis are then summarised. The implementations focus upon demonstrating that the visualisations may be generated automatically with minimal human intervention, and so less important features are not included. In addition, the process of obtaining and processing the required data for these tools is also explained, to show that this stage may also be automated.

6.2. Quack

Animation is a key concept of both Revision Towers and HfVis. Therefore, it was determined that the implementation should be built upon a dedicated animation system, rather than adding animation facilities to an existing visualisation toolkit.

The choice of available animation systems was very limited. Algorithm animation systems such as Tango [Stasko90] were considered and rejected. The main problems with such systems were that the animations that could be produced were limited in scale, and that the systems were often designed for use with handcrafted animations. Also, the systems were out of date with respect to the graphical abilities that could be achieved with more modern technology, such as high colour displays, antialiasing and transparency.

The other option was to use a tool such as Flash [Flash03]. Flash is a modern, dedicated animation tool, with excellent graphical support. In addition, it has good support for interactive animations, is widely available, and is relatively fast. However, at the time of implementation, Flash had two significant problems. The first was one of scale, with the performance deteriorating significantly when a large number of objects were introduced. The second was that there was very little support for dynamically generated animations, as Flash is designed for creating specific animations through a dedicated graphical front end. As automation was an important aspect of the visualisations to be created, Flash was therefore considered to be unsuitable. A small number of libraries also exist that will produce output suitable for Flash from a standard language such as C++. However, these libraries were found to be incomplete, with the output also of a low quality.

Given this situation, it was decided that a new animation system would need to be developed in order for the subsequent visualisations to be successful. Quack (**qu**ick **a**nimation in **c**++ **k**it) was designed and implemented as a mid-level animation system. The cross-platform open source graphics library Allegro [Allegro03] was used as the basis for this system. Allegro was originally developed as a highly optimised low-level games programming library, and is particularly suited to development requiring two-dimensional graphical capabilities.

6.2.1. Features

Quack is based on a key frame arrangement. The central concept within the system is that of an animated object. At each key frame, each object may be given a different location, shape and other properties. Therefore, the same object may be represented as a red square at one point in the animation and then a green triangle at a later point. Within the visualisations, an animated object is designed to represent the same underlying component throughout the animation. As the functionality or behaviour of the component may change, it was necessary to be able to handle this within the animation system.

The animation itself is programmed using a series of blocks, each containing a number of animated objects or further blocks. Each block has a number of properties, and may also be key framed. The properties of a block also propagate down to further blocks and objects. This provides a grouping mechanism, allowing properties of several objects to be changed simultaneously. Perhaps the most useful benefit of this is the ability to move the contents of a block by changing the co-ordinates of the parent block. An object, or block, may also be attached to more than one block, allowing blocks to be repeated across the display with different properties.

Smooth motion is an important part of a system. Every property of a block or object will be smoothly interpolated when in-between frames are calculated. In addition, each property of each object may be provided with a different interpolation function. This provides a large amount of flexibility that is difficult to achieve in other systems. For example, a change in colour may be provided with a linear interpolation function. At the same time, the co-ordinates of the object may be changed using a sinusoidal function instead, to provide a slow-in slow-out motion. The co-ordinates of the object could also be converted to polar co-ordinates by the function, allowing radial, instead of linear, motion. More complex paths may be created in a similar way. In order to simplify the construction of this part of the animation, a number of default interpolation functions are included.

Although animations are programmed in advance, the programming may be changed as the animation is playing. This allows interaction operations to have an immediate effect on the output of the animation. In addition, although optimised for animations to be played in a forward direction, Quack supports jumping to any frame in the sequence. This then allows animations to be played in reverse, and also for frames to be skipped for faster playback.

A number of additional features are also included to provide better support for developing visualisations with the system. Picking, or the ability to determine the object at a given location, is an integral part of Quack. This information is updated in real time so that the object under the mouse pointer may be determined even whilst the object is moving.

Filters are related to this feature, and may be applied to blocks and objects on a key frame basis or instantaneously. A filter acts as if a coloured filter was placed over a lens, and may change the hue, saturation or lightness of colours in the affected blocks on a relative or absolute basis. Filters may also be combined. Using filters together with picking means that objects can be highlighted easily to show

they have been selected. Filters may also be used for highlighting the results of a query, or to fade objects in and out of the display.

Within the animation, each object and block may also be allocated a different depth. This ensures that objects may be overlaid and cross over in a predictable fashion. Combining this functionality with filters allows other features that are not standard in a 2D system, such as darkening distant objects to simulate fog.

Finally, Quack supports smooth zooming, which may be activated simultaneously with any other motion. A fractional co-ordinate system is used in order to provide pixel perfect placement of objects even at high zoom levels. In addition, each block may have a number of different blocks or objects attached, depending on the current zoom level. This provides level of detail functionality to be implemented at a number of different levels, for example by providing a less detailed version of a single object or an entire group. The number of alternative zoom levels is unlimited.

6.2.2. Limitations

Although many features are provided by Quack that are often requested within visualisations, a number of features are not yet available due to the necessity of the research. In particular, the animation system has been optimised for speed, and therefore features have not been included that would have a detrimental effect on this.

The most significant feature missing is support for a windowing system, and associated graphical user interface. Quack provides a vastly superior frame rate of the order of 300% when it is executed full screen, rather than in a window. However, this prevents the native GUI of the operating system from being used effectively. Although Allegro provides a very simple GUI, it was not thought to be sufficiently flexible to be used. The effect of this is that the interactive aspects of a visualisation are much more difficult to implement. In particular, search and query operations are difficult to implement, as common widgets for data entry and parameter selection are not available. Similarly, although multiple views may be shown in fixed areas of the screen, there is no support for resizing these views through a user interface. Floating windows are also possible, although the library is not optimised to handle these efficiently and so the available frame rate will be significantly reduced.

6.3. Revision Towers

Revision Towers, as described in chapter 4, has been implemented as a proof of concept tool. The tool is implemented in C++, using Quack as the underlying graphics and animation engine. The purpose of the conceptual tool was to demonstrate the feasibility of the idea when data was taken directly from currently developed open source projects, rather than using artificial data. In addition, it was important to investigate whether the visualisation could be successfully automated.

6.3.1. Overview of process

The implementation of Revision Towers uses log files provided by CVS as the sole data source. CVS was chosen as the vast majority of open source projects use this for configuration management, and so it allowed access to the large amounts of historical data required.

Useful data is obviously lost by restricting the data provided to log files only. However, there are two major benefits. The first is that the data is easily accessible, and may be retrieved directly from the CVS server. The second is that the computational effort that is required to further process the log is minimal. This therefore provides a good balance between the quality and quantity of the information that may be retrieved from the visualisation, and the expense of generating that data.

For the proof of concept tool, a four-stage process is required for generating the visualisation for a new open source project, as demonstrated in Figure 6-1. This assumes that a local copy of CVS is available to the user.

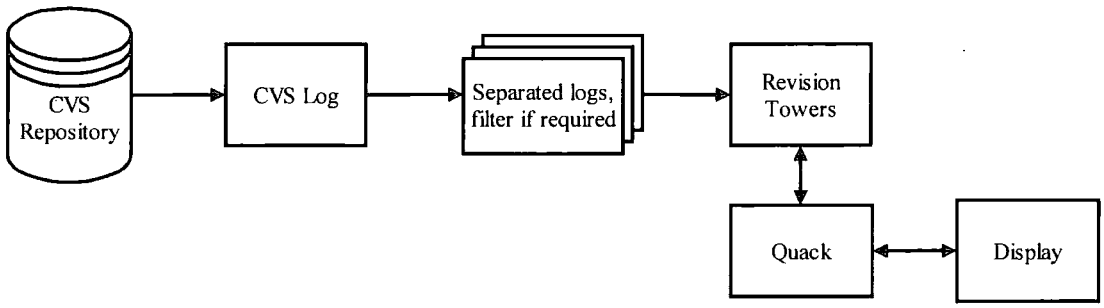


Figure 6-1. Revison Towers process.

1. Connect to the CVS server, and retrieve the latest version of the project. For example, to obtain Allegro the command is:

```
cvs -z3 -d:pserver:anonymous@cvs.alleg.sourceforge.net:/cvsroot/alleg
co allegro
```

2. Generate a large log file containing the details of all of files within the project. This may be achieved with:

```
cvs -z3 -d:pserver:anonymous@cvs.alleg.sourceforge.net:/cvsroot/alleg
log > logfiles.txt
```

3. Separate this log file into individual logs for each file, using a simple script. Preliminary filtering may be done at this point to reduce the number of files provided to the visualisation. For example, a single module of the project may be viewed by limiting logs to those of files contained in a particular directory on the server.

4. Start the visualisation providing the directory as a parameter.

These stages could be simplified in two ways. Firstly, the individual log files could be provided as part of the project documentation, and be downloaded separately. This removes the need to install CVS and

obtain the project, and may be suitable for those considering using the project as an end user, rather than participating actively in its development. The second solution involves integrating CVS and Revision Towers together, removing the need for any pre-processing. The extra implementation required to achieve this was not considered to be worthwhile within the proof of concept tool, although a complete system should use this second approach, as demonstrated in Figure 6-2.

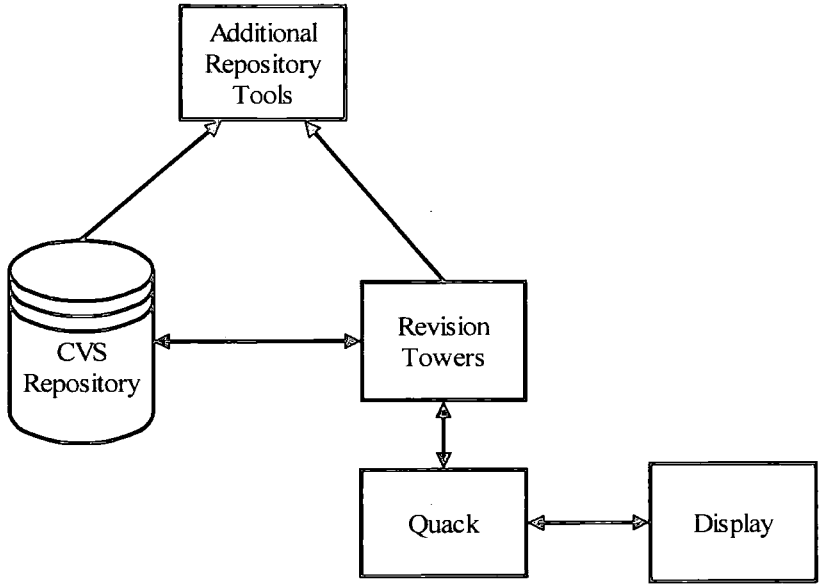


Figure 6-2. Ideal Revision Towers process

6.3.2. Limitations of the tool

Some aspects of Revision Towers have not been completely implemented because of the restrictions in the implementation of Quack, the underlying animation engine.

Maintaining a high frame rate was considered to be important, and so Revision Towers is run in full screen mode. The effect of this is that no graphical interfaces could be provided. Similarly, floating windows were also unavailable. In order to counteract these issues, the visualisation is controlled entirely from the keyboard, with the mouse used only to select objects within the display. Instead of floating windows, small overlays are displayed to show the timeline and the author colour key. These overlays are translucent in order to highlight that they are covering potentially important data. In addition, overlays may be moved to a number of fixed positions or removed completely, again to avoid obscuring data.

In addition, due to the restricted interface, an accessible implementation of a query language to select items within the visualisation was not provided. Some of the other interactive events within the visualisation were also not included for this reason. Additionally, in order to simplify the implementation of the animation, complete towers are generated and partially pre-rendered. Filters are then applied to this animation at various points during play back to provide fading and highlighting operations. However, this limited the available interactive operations that could successfully be applied within this framework. An implementation of a table lens as a focus and context view has already been

realised by others, and so it was not re-implemented. Grouping and branch expansion were other notable features that were left out due to the partly pre-rendered nature of the animation.

Finally, the tool is a stand-alone visualisation and has no support for integration with other tools. This is as a result of driving the visualisation from generated log data, rather than retrieving this data directly from the server. As there is no direct connection between the visualisation and the configuration management server, checking out files or generating diff information within the visualisation is not possible.

6.4. HfVis

A proof of concept tool of the HfVis visualisation outlined in chapter five has also been implemented. The tool was again developed in C++, using Quack to provide animation support. The focus of the tool was to determine whether the idea could be automated successfully, using data retrieved from current open source projects.

The tool was restricted to using data provided from the fact extractor, rather than also including data from the configuration management system. The reason for this is that much of the configuration management data required could already be retrieved from the Revision Towers conceptual tool, demonstrating that it was feasible to automate this part of the visualisation.

6.4.1. Overview of process

The fact extractor chosen for the implementation of the tool was 'Doxygen' [Doxygen03]. Doxygen is an open source document generating tool developed in C++, with similar behaviour to JavaDoc [JavaDoc03]. The tool takes a series of files as input, and will output these files as a series of HTML pages containing class or function details, with hyperlinks used for easy cross referencing. Additionally, documentation contained within the code is also inserted into the pages.

The decision to use a documenting tool, rather than a dedicated fact extractor such as Datrix [Datrix03] or Columbus [Ferenc02] is perhaps unusual for a tool of this nature. However, there are a number of reasons why such a tool, and Doxygen in particular, was chosen.

Firstly, it was decided that in order for the visualisation to be used in practice, it would have to be integrated into the open source process. Perhaps the poorest aspect of open source development is the difficulty of maintaining up to date documentation. In order to counteract this, some open source projects will embed the documentation within the code, and this can then be extracted later using a documentation tool when the release is made. Therefore, by using the same documentation tool to generate the data required by HfVis, no additional steps need to be added to the process.

The second advantage is that some of the pre-processor issues inherent in parsing C and C++ source code are avoided. Documentation tools will allow some #defines to be provided when generating the documentation. These are necessary when macro expansions are required in order to parse the code successfully, or if the user wishes to generate documentation for a specific set of #defines. Assuming

that the settings have been set correctly to generate the documentation successfully, these same settings may then be used to also generate the data required by HfVis. In addition, the documentation and visualisation will remain consistent, allowing further opportunities for cross-referencing between the two.

Finally, Doxygen was chosen over similar tools for a number of reasons. Firstly, the tool allows the data generated internally to develop the documentation to be exported as XML. It is this XML that is then used by HfVis, rather than further parsing of the HTML files produced. Secondly, although originally intended for parsing C and C++, the tool has being extended to work with other languages common within open source, such as Java, Perl and PHP. The output from these languages is very similar to that for C and C++, and so this would allow HfVis to be modified easily for use with other languages. Finally, the tool is still undergoing active development with regular fixes, enhancements and optimisations.

Doxygen does have two main limitations however. The first is that only partial call graph information is available, with only a subset of the static data provided. No dynamic call information is available at all. Therefore, the proof of concept tool does not provide any call graph information. The second is an issue common to all available fact extractors, in that they have been designed to generate facts for a single release. Therefore, the complete source code for that release must be provided to the fact extractor for every release that is to be visualised. Similarly, complete output is also given for every release, rather than an incremental update from the previous release. As the parsing and generation process is the most expensive aspect of the visualisation, reducing the effort required would reduce the overall resource requirements of the visualisation significantly.

A four-stage process is required in order to use the proof of concept tool, as shown in Figure 6-3. The process assumes that both CVS and Doxygen are available.

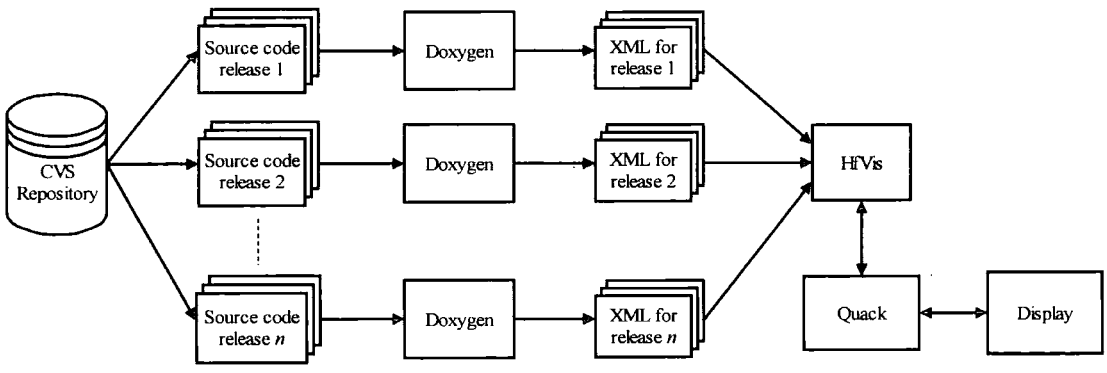


Figure 6-3. HfVis process diagram.

1. Retrieve the complete source code for a release to be included in the visualisation.
2. Run Doxygen, giving the source code as input, and a release-specific directory for output. This will generate all of the XML files containing details of that release.

3. Repeat 1 and 2 for every release that is to be included.

4. Start the visualisation. Initially, this will parse the XML to recreate cross reference information, match files and components of files across releases, and determine new and deleted elements. The force directed view will then be displayed.

As with Revision Towers, the process may be simplified by generating new data after every release as part of the release process, and making this data available to the user. Integrating the CVS aspects into a single stage is less appropriate than within Revision Towers, due to the additional fact extracting stage that is required.

6.4.2. Limitations

As with Revision Towers, maintaining a high frame rate was considered paramount. This necessitated the use of Quack in full screen mode, and so again severely limited the available user interface. In particular, the diff-based view was not implemented, as several tools exist to provide this functionality already. For example, the free tool CsDiff [CsDiff03] provides clear and concise output, and could be easily integrated if HfVis was run in a window rather than full screen.

In order to simplify the implementation of the animated parts of the visualisation, the animation is pre-rendered in chunks corresponding to each release. Therefore, any structural changes required, such as switching between views or levels of detail, must be made at the end of the section. Due to the limited user interface available a number of the interaction methods detailed, such as querying, were also not implemented.

6.5. Summary

This chapter has provided a brief overview of the implementation of the visualisations Revision Towers and HfVis detailed in the previous chapters. In order to increase the flexibility and efficiency of the animations that are produced, each tool is based a custom graphics and animation engine, Quack. The process of generating the visualisations from the version control repository, and some of the design decisions influencing this process, has also been described.

ANIMATING THE EVOLUTION OF SOFTWARE

7. Evaluation

7.1. Introduction

This chapter presents an evaluation of the two visualisations detailed in chapters 4 and 5. There are a number of techniques that may be used for evaluating visualisations of this nature, and these are summarised. Following this, justification for the use of the chosen methods is provided.

The visualisations will then be evaluated separately using both formal and informal methods. Subsequently, the differences between these visualisations and existing systems will then be shown, in terms of the ability to view the evolution of the software. Finally, a number of examples of the use of the visualisations will be provided. Neither visualisation has been subject to industrial trials.

7.2. Evaluation Methods

There are a number of alternative techniques for software visualisation evaluation. Some of the options available have been set out by Hatch et al. [Hatch01], based mainly on software engineering evaluation techniques by Kitchenham and Jones [Kitchenham96]. These options will be summarised together with some of the benefits and pitfalls of using them.

Evaluation frameworks are a popular evaluation mechanism, as they may be applied easily. No prerequisites are imposed on either the environment or the target system, allowing the strategy to be employed in a large number of cases. They are particularly valuable, however, for assessing feature rich systems, as the benefit of a single feature within the system may be identified. Finally, the multiple-choice nature of many frameworks means that comparisons with different systems may be made with a low investment of time.

Despite the apparent simplicity of frameworks, there are a number of issues that must be considered. A simple yes/no question within the framework is open to misinterpretation, as the requested feature may be provided in a manner not intended by the author. However, a question with a sliding scale may become too subjective, and rely on the evaluator's background and experience of the visualisation tool. Including less quantifiable features, such as the extent to which Gestalt effects distort the mental view of a user, is also difficult, and therefore rarely considered. Finally, the limited number of available frameworks will often mean that the author of a visualisation may create their own dedicated framework. This can lead to a process of self-evaluation.

The second method is the use of feature analysis. This involves a process of identifying high level features that are in some way important to the task for which the visualisation is required. Once identified, individual visualisations may then be examined to determine the extent to which these features are supported. A score may be generated for each visualisation by summing the features, with the important features required given greater emphasis in the final score through the use of weightings. The disadvantage of this approach is that it only indicates the features that the visualisation supports, and provides no information as to how easily the features may be used, or the extent to which the features may be used simultaneously in order to solve the given task.

Scenarios represent an alternative evaluation method that demonstrates the features of a visualisation in practice. The burden of the visualisation is then moved to the readers, who may evaluate the tool according to their own requirements. Scenarios will often present the visualisation in terms of a problem, and the process by which the solution may be found. This will be done using a combination of textual descriptions and screen shots, allowing the reader to follow the process clearly.

Again, scenarios present a number of problems. The author is in full control of the information to be presented, and thus may choose to describe and highlight only those features that are executed well by the visualisation. A further issue is that, by their very nature, scenarios are verbose and so only a limited number may be presented in any detail, allowing less favourable scenarios to be excluded easily.

A further option is the use of empirical studies, which may be carried out in order to allow statistical analysis of the success of the visualisation. This allows claims made of the visualisation, such as that it reduces the number of errors made in a task to be verified. User studies will also record individual and collective feedback on the perceived ease of use and benefits of the visualisation.

However, such studies are subject to many difficulties. In order to be effective, a stable and complete implementation of the visualisation is required. Additionally, the subjects and tasks chosen for the study will affect the final conclusions that may be drawn, as it is not always possible to apply the experimental results to a real-world scenario. Therefore, to draw accurate conclusions, professional developers would be required to use the visualisation on large software with complex tasks, which has significant resource implications.

Finally, the visualisation may be evaluated informally, by critically examining a number of the individual features of the visualisation. This is obviously the most subjective of the available methods, and is wholly dependent on the individual opinions of the author. However, the approach does allow many of the aspects of the visualisation that are not examined specifically using the other techniques to be evaluated.

Of these, four different techniques have been used to evaluate Revision Towers and HfVis. The use of evaluation frameworks allows the key visualisation concepts of the tools to be evaluated. A broad feature based analysis is also used, in order to highlight the similarities and differences between these visualisations and similar systems. Scenarios are also used in conjunction with these methods to demonstrate how the visualisations may be used to solve specific software engineering tasks, rather than just stating that the visualisations have theoretical support for generic tasks. Industrial trials were not used for either visualisation, due to the requirement of complete implementations rather than proof of concept tools. However, both visualisations were reviewed informally, with positive feedback.

In addition, the visualisations will be evaluated informally. This evaluation will critically examine issues regarding the use of the underlying data, benefits and drawbacks of the layout and representation used, and the value of using animation as part of the visualisation. Animation adds a further dimension to the visualisation and so increases complexity, and therefore this increased complexity must be

justified. Finally, the resource implications associated with empirical studies meant that an experimental approach was not feasible.

7.2.1. Description of frameworks

A very small number of evaluation frameworks exist that are suitable for Revision Towers and HfVis. Both of these visualisations are tailored towards examining evolving data, and using animation within the representation as a means of achieving this. Unfortunately, current frameworks do not consider all of this information specifically. However, creating a new framework to target HfVis and Revision Towers would generate problems of self-evaluation. Therefore, existing frameworks have been used as a basis for evaluation of these visualisations.

Of the frameworks in existence, those of Storey et al. [Storey97] and Knight [Knight00a] are perhaps the most relevant, and have been used as a basis for evaluating a number of other visualisations. Storey's framework is designed for "graphical representations of static software structures linked to textual source code", which they refer to as a software exploration tool. The aim of such a tool is to help a maintainer form a mental model of the software, in order to improve the understanding of the code before modifications are made. The framework is split into two separate parts, which examine the role of the tool in improving program comprehension, and whether the cognitive overhead of the maintainer has been reduced, as shown in Figure 7-1.

Knight's framework builds on this framework by introducing some additional aspects that are not covered by Storey et al. In particular, a generic visualisation section is included that is not restricted to software exploration tools. The purpose of this is to combine the task support provided by the tool with an indication of the extent to which the user will be able to extract the information required from the visualisation.

A combination of these two frameworks will allow many of the aspects of Revision Towers and HfVis to be evaluated. However, it must be recognised that the tools do not fit neatly within either framework. Software exploration tools that existed when the Storey framework was developed were based on deriving relationship details of modules, objects and functions from source code for a single software release, and then visualising these relationships. Neither Revision Towers nor HfVis consider this information as the primary data source, and instead concentrate on the change history information that is available.

However, that is not to say that neither tool may be classed as a software exploration tool. Revision Towers allows implicit relationships to be seen that are not necessarily obvious from the source code alone. Demonstrating how and when files have changed, and showing which authors have been involved in those changes may identify related files if they were modified simultaneously, or by the same author. In this case, the textual source code referred to in the definition of a software exploration tool may be considered to be the raw log information provided to the visualisation, and the software structures as the changes made to a file. Additionally, very simple relationship information is also available by linking header and implementation files together.

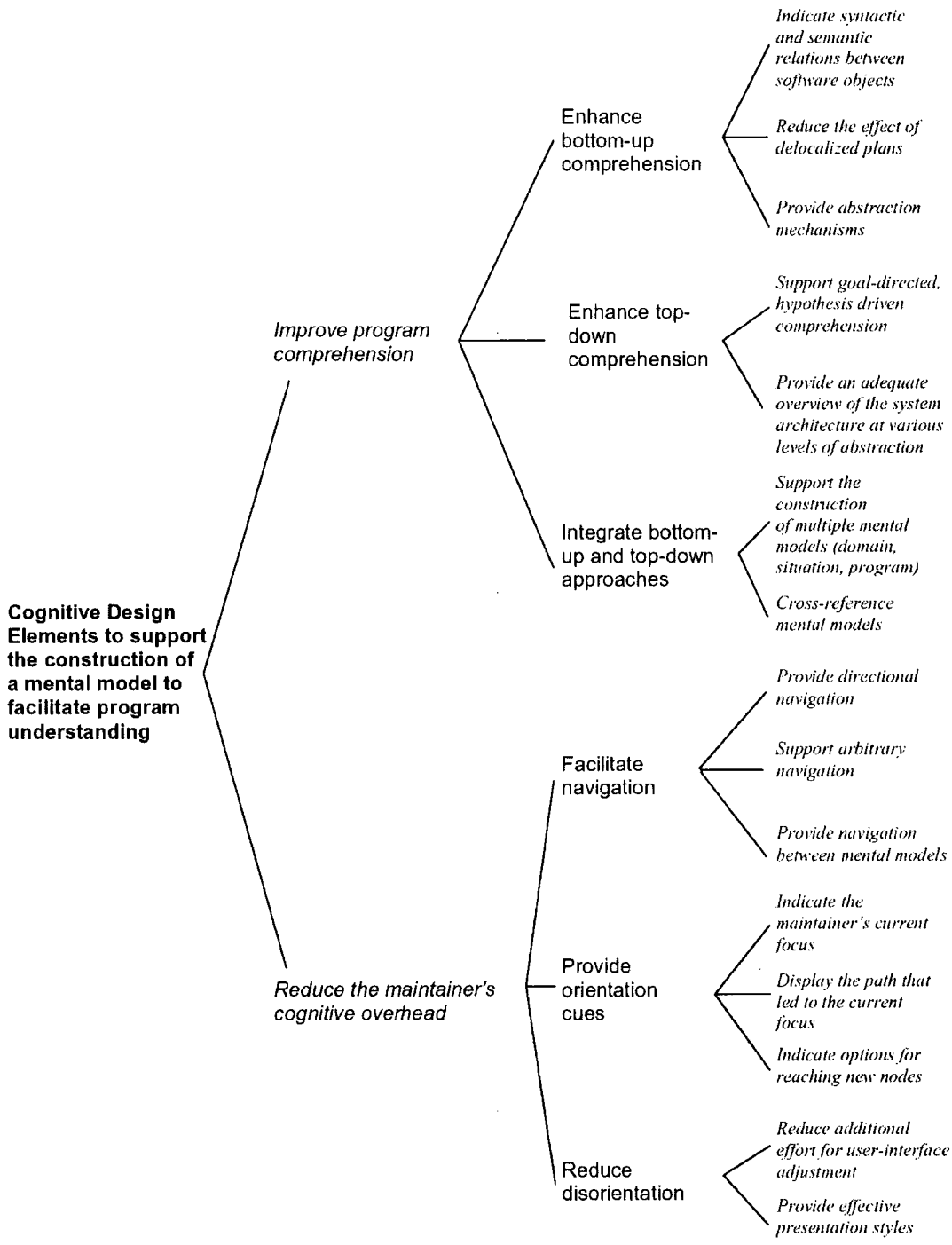


Figure 7-1. Evaluation framework for software exploration tools [Storey97].

HfVis is a more obvious example of a software exploration tool. Here, the main relationship information provided by the visualisation is derived from the source code. The exact relationship details that may be provided are dependent on the fact extractor used to obtain the data. Therefore, although neither HfVis nor Revision Towers focus uniquely on the program comprehension aspects of software evolution as considered by Storey et al., both may be considered as a software exploration

tool. However, it is important to recognise that the framework considers only static data structures, and so there will be several aspects of the tools not covered by the framework.

The framework by Knight et al. is also appropriate for HfVis and Revision Towers. Again, the framework was developed from a program comprehension perspective, rather than a more general software evolution viewpoint. Additionally, the visualisation aspect of the framework was developed to also consider specific issues that were raised when developing 3D visualisations. Therefore, there are aspects of this framework that are not applicable to HfVis and Revision Towers. However, in many cases, it is possible to reapply the precise questions posed within the framework. For example, static screen displays of both animated and 3D environments may result in misunderstanding, as the context of the display may not be obvious. Therefore, by understanding the rationale behind the original question, it is possible to reapply the question safely to a new domain with less danger of self-evaluation than if a completely new framework was developed.

It must also be recognised that Revision Towers and HfVis place a much greater emphasis on the issues inherent within software evolution than is considered by either framework. Although Knight considers evolution to a very small extent, the additional issues raised when evolutionary data is considered, such as how layout and representation issues may be resolved, are not addressed specifically. Similarly, as the frameworks were developed around a static program comprehension domain, neither framework includes any issues relating to change history, or even configuration management, which are fundamental to the development of Revision Towers and HfVis. Finally, both tools use animation as part of the visualisation mechanism. Although it is possible to reapply some of the 3D visualisation questions to animation, animation introduces separate issues that are again not covered by the framework.

However, developing a specific framework for these issues would be likely to result in a high level of self-evaluation. These issues will instead be evaluated informally, and also through the use of scenarios.

7.3. Revision Towers Evaluation

This section will serve to evaluate Revision Towers. Initially, an informal evaluation will be undertaken, examining a number of issues specific to Revision Towers that may not be covered by the other methods. Secondly, the visualisation will be evaluated using the frameworks mentioned previously.

7.3.1. Informal evaluation

Before a formal evaluation of Revision Towers is undertaken, some of the aspects of the visualisation will be examined informally. The purpose of this is to provide details, and examine issues specific to this visualisation, rather than only applying a more generalised framework. Four areas will be considered in some depth; the data source used within the visualisation, features of the representation used, the effectiveness of the layout, and the benefit of adding animation.

7.3.1.1. Data Source

Revision Towers was developed to show the raw log data contained within a typical version control system in a meaningful manner. However, it is important to investigate the extent to which this data is used effectively by the visualisation, and whether additional data is required.

Use of a static data source

The log data contained within a version control system will change on a very regular basis. The speed of this change will depend on a number of factors such as the project size and popularity, and the current position of the project within a release cycle. However, whilst recognising that the data presented to Revision Towers will evolve, the visualisation also treats the data as a static data set. In other words, the latest log may be provided as input to the visualisation. Once the visualisation is then viewing this data, any other changes made to the log will be ignored until the visualisation is restarted with the evolved data set.

The extent to which this is a problem is dependent on the target audience. If the visualisation were to be incorporated into a web-based environment intended primarily for developers and end users, the use of static data would present few difficulties. The visualisation could be included in a similar way to an existing text based view of the log, where the information is generated each time it is requested. As the visualisation requires minimal computation, this would be feasible for Revision Towers. A web-based view is also likely to be used for short periods of time, and so the probability of a large number of changes affecting the project significantly will be very low.

However, if project managers were to use the visualisation to monitor the state of the project on a continual basis, then static data would be less appropriate. In this case, it is envisaged that the visualisation would be a stand-alone application that would be running continuously. A direct connection from the visualisation to the version control system would be required, and new data would then be retrieved from the system whenever a change occurred. The visualisation would then need to update itself to include the new data.

At the simplest level, this could be achieved through a simple refresh operation where the visualisation is effectively restarted whilst maintaining the current state such as timeframe and query results. However, if the changes involved additional files or releases, then the impact on the display would be greater, with slight resizing or relocation of most towers required in order to incorporate the new data. This would best be achieved through an additional item in the user interface, indicating that the project data has changed. When this item was selected, or after an optional time-out period had expired, animation would then be used to morph the old display into the new. As the layouts of the old and new displays would be very similar, the morphing operation should maintain a high level of context. The human intervention required to update the display should avoid movement created by the animation distracting the manager from any tasks they were involved with, as they can delay the refresh until the task is complete.

Effectiveness of data

Revision Towers uses a version control log as the primary source of data. This information source is very accessible, and much of the data is simple to extract due to the fixed format nature of the log files. The obvious benefit of this is that minimal computation is required when generating the visualisation. Additionally, this initial data is available automatically with no human intervention, making Revision Towers practical for both experienced and inexperienced end users.

However, although Revision Towers presents a useful visualisation of this data, it is also important to evaluate how much may be determined about the project from just examining this limited data set. The scenarios that are included later in this chapter will demonstrate some of the inferences that can be made about a project using Revision Towers, from overviews of the project to areas where re-engineering may be required.

However, there is some significant data that has not been included within the Revision Towers visualisation. Mailing lists, newsgroups and forums provide a wealth of information, discussing design decisions and possible solutions. Fault reports may provide the reasons for the submission of a subsequent patch. Although this data would enhance Revision Towers significantly, extracting data from these sources reliably and accurately is not currently feasible. Even making the data accessible at a particular point within the visualisation would be difficult. For example, it would be useful to be able to display release notes or a high-level change log when a release was selected within the visualisation. However, such an operation would be different for every project, and also assumes that the format of the notes remains the same within that project. Similarly, selecting a period of time on the timeline could display mailing list archives during that time. Again though, this assumes that the archives are available to the visualisation, and may be extracted in a consistent format.

Realistically, the most viable approach for Revision Towers would be to provide access to this information as part of the user interface, rather than attempting to visualise this accurately. The data could then be accessed externally, using existing tools such as archive search engines. These tools could still be integrated into the visualisation though, allowing parameters to be passed representing a particular release or timeframe, for example.

Revision Towers presents a suitable starting point for investigation of the project, and may provide some probable answers to questions that may be posed. However, the use of a log as a data source can not provide the full story, and the visualisation is best used in conjunction with these other data sources in order to provide the user with the most accurate image of the project.

7.3.1.2. Representation

A single Revision Tower is a simple representation, which has a low learning curve for new users. However, there are a number of issues regarding this representation that need to be evaluated further.

Relative sizes

Firstly, relative sizes are used predominantly within Revision Towers, rather than absolute measures. The layout algorithm means that each tower must be displayed in a fixed space. Therefore, in order to restrict the tower to this space, it is obviously not possible to have a 1:1 mapping between file size and pixels used within the representation. A project wide ratio, where the same ratio is applied to every tower within the visualisation, means that a single large file size will impact upon every small file, making them difficult to read, and create the potential for changes being missed. The use of relative sizes avoids this issue.

However, relative sizes create other difficulties, particularly when comparisons are made between small and large files. The initial impression from examining the two towers in Figure 7-2 side by side is that both towers are the same size, and have undergone the same amount of change. Although the line at the base of the towers represents the scaling factor, and so indicates that this is not the case, the first impression is difficult to dispel.

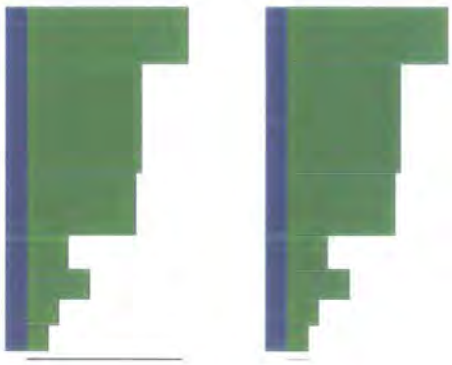


Figure 7-2. Two towers, showing very large and very small files.

This could be improved through the use of a logarithmic function to determine the maximum width for each tower. This would ensure that there was a greater visual difference between small and large files, whilst also ensuring that small files were not swamped. Having determined the maximum width, the individual sections would then be allocated using a linear mapping, as before. This allows comparisons within the tower to be made more easily than if a logarithmic mapping were used. Figure 7-3 illustrates this new layout, showing that the file displayed in the right tower is smaller than that in the left tower.

This solution is still far from perfect, as it is still difficult to prevent instant comparisons being made. For example, it now appears that the file that is shown in the right tower is now half the size of the file that is shown in the left tower, when in fact this is not the case. However, it does emphasise that the files are of a different size.

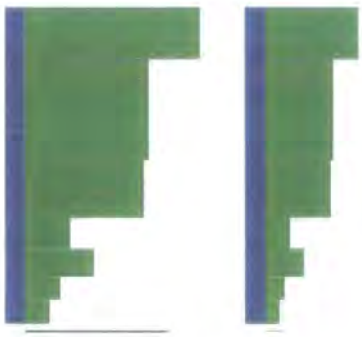


Figure 7-3. The same towers, with the maximum width related to the overall file size.

Underlying assumptions

A Revision Tower is based on the underlying assumption that releases will tend to incorporate the most recent version of a file. This is particularly the case when the project is smaller, and branching is not often used for experimental development. The impact of this is that a specific version of a file will not be attached to more than one release unless those releases are sequential. The representation will handle this latter case by resizing the version to span multiple releases.

However, there may be projects where this is not the case, and a specific version is repeated at numerous locations. Such a situation may arise if a new feature was introduced that was then removed after a short time, for example, if the feature was later found to be insecure and had to be patched quickly. Provided this occurs infrequently, a line may be drawn between the two identical versions within the tower indicating that the second version is a repeat of the first, without any impact on the rest of the tower. If the situation should arise on a very regular basis though, the number of connecting lines required will be significant. It is probable that these will also have a high number of crossings, making the tower very difficult to understand. Therefore, because of this underlying assumption, Revision Towers will perform poorly on projects where this assumption does not hold for the majority of files.

Use of colour

Colour is a critical attribute of the representation used within Revision Towers. The important benefit of colour over other visualisation attributes such as size and shape is that a change in colour will not have any layout implications. This is of particular value in Revision Towers as the layout is very restricted in order to incorporate the future proofing aspect of an evolutionary visualisation. Possible alternatives, such as the use of a 3D, rather than a 2D, representation were rejected, as the combination of 3D and animation would have created other issues of complexity and occluded changes. The use of different patterns as an attribute was also rejected, as some patterned sections were too small to be able to distinguish the patterns successfully. Patterns are also thought to increase visual complexity significantly [Tufte83].

The reliance on colour means that the visualisation has accessibility implications, as users with colour blindness may find they are unable to use all of the functionality within Revision Towers. Colours are also overloaded, resulting in increased cognitive complexity. Specifically, colours in the central section of a tower need to be interpreted differently from colours used in the side sections of a tower. The colours have been selected so that the overloading is reduced, with the central section using far darker colours than those used for the side section, although this will not always apply when zooming and abstraction techniques are required. By grouping authors together, for example, by ensuring that authors making a small number of contributions are given a single colour, the number of colours required is also reduced.

It is possible for the overloading to be reduced further. If the visualisation were restricted so that every tower had to use the same vertical mapping for releases, colour would not be required for the central section. However, this reduces the flexibility offered, and assumes that every file within a project has evolved in a similar manner. Therefore, this functionality should be optional.

7.3.1.3. Layout

Revision Towers uses a chronologically ordered grid layout in order to display multiple towers. The objective of this layout is to reduce the change in layout required when the data evolves, both during the animation phase of the visualisation, and externally when new files and versions are added to the project.

Fixed size towers

Each tower within the visualisation has the same initial width, and under most display modes, the same height. Therefore, the space allocated to a tower that has existed for many releases, and which is likely to contain much significant data, is the same as the space allocated to a new tower containing a single file. If the intention was to use the visualisation only once for the project, then this would be an unacceptable use of the available resources, and a different layout would be required with a more optimal use of the available display space. However, instead it is important to consider that the visualisation will continue to evolve, and that the space allocated to a tower will become filled at some point in the future. It was also decided that a high level of consistency within the layout was more important than an optimal placement.

Chronological ordering

The ordering method used is an important part of any fixed layout. The default method used in most situations will be an alphabetical ordering, such as in Software World [Knight00a]. Alphabetical ordering is also used with online open source repositories to view files through a web-based interface. Comparatively, chronological ordering is very rarely available in such environments. However, Revision Towers uses this ordering in order to provide a greater degree of future proofing within the visualisation. Therefore, the drawbacks of this approach must be considered.

Firstly, the use of such an ordering will be unfamiliar to an end user, and a slight cognitive overhead will exist as the user translates between the usual alphabetical ordering and the chronological ordering used in the visualisation. However, other than this familiarity, there is little benefit in using alphabetical ordering. The usual advantage of alphabetical ordering is that it enables a specific, known entity to be found quickly amongst a large list. This is not an important aspect of Revision Towers. Instead, it is envisaged that an area of interesting or unusual activity will be identified within the visualisation, and then the files involved in this activity will be identified. The ability to locate a specific file on demand may instead be included as part of the interface. More interesting patterns will also be identified as part of a chronological ordering, where the behaviour of files that have existed for a long time may be compared easily to code added recently.

Secondly, alphabetical ordering may also indicate related files. This may occur if the project uses a specific naming scheme, with prefixes used to identify files which exhibit the same behaviour or which are part of the same module. If names are not shown to save display space, a user may then reasonably consider that two adjacent files are related. Within Revision Towers, files related in this manner may be shown with an additional identifier after the filename. Combining this with suitable filter operations allows related towers to be identified and compared in a similar way as for an alphabetical layout. However, only one such group may be examined in this way, with multiple group-based comparisons more difficult to achieve.

Big-box layout strategy

Revision Towers uses a big box layout strategy (see section 3.4.1.2). The technique relies on leaving areas of space unallocated during the initial layout algorithm. The space will instead be allocated at a later date in the evolution of the system. A balance must then be struck between leaving very little unallocated space, and so allowing the existing elements within the visualisation to cover most of the display, or leaving a lot of unallocated space, and so reducing the probability of running out of space at a later point. Revision Towers leaves a constant amount of space unallocated, in order to provide significant space to display the existing elements.

However, Lehman has shown that the number of files within a project over time maps onto an inverse square curve. Therefore, more space should be provided at the start of the project than the end. Although the extent to which this applies within open source software development in general is not yet known, the constant allocation method used by Revision Towers is not optimal. Instead, the layout algorithm should become more intelligent, and analyse the data set in order to determine whether there is a trend in the number of files being introduced in each release. The space allocated should reflect this trend, so less space is reserved if the number of files being introduced with each release is decreasing.

7.3.1.4. Animation

The final aspect of Revision Towers that will be evaluated is the use of animation. Revision Towers uses animation for a number of operations. However, no additional information is provided by the animation that is not available through other means within the visualisation. Therefore, it is important

to examine to what extent the animation is necessary, and whether the visualisation would suffer if the animation were removed.

The main operation where animation is used involves viewing the changes in the project over time. The timeline indicates the current position within the timeframe of the project, and parts of the towers fade in at the appropriate points to represent the current state at that time. One of the benefits of animation for this task is that the user is not immediately presented with the complex picture displayed at the end of the timeframe. Instead, the visualisation is built up slowly, with the speed controlled by the user.

The user may also be alerted to areas of change within the project during this process, although the extent to which this will arise depends upon the focus of the user. If they are examining a specific area, then change blindness research indicates that changes elsewhere may not be noticed. If instead the user is focusing on the entire display in order to obtain an overview of the project, changes are more likely to be noticed. Therefore, animation is also relevant for this task, as the user may observe changes that would otherwise go unnoticed if a single picture were presented.

The final aspect of the animation is that new versions will fade in at different points in time, corresponding to the actual dates contained within the log. This is useful for a single tower, as this may highlight the project process. For example, a version with a long time frame followed by many quick versions may indicate significant development followed by fault fixes. It is also possible to estimate whether other versions in other files are being updated at the same time, which may indicate that the files are related. However, unless these towers are in close vicinity, it is difficult to make this comparison accurately, as the movement of other towers may be distracting. In this case, queries should be used to identify the exact files.

Therefore, the use of animation for viewing project changes does provide significant, but limited, benefit for Revision Towers. The main advantage is that the user may be alerted to areas of interest, which may otherwise be missed in the static view. However, the animation phase is too weak to be able to draw definite conclusions. Instead, once the user is aware of a possible relationship, other methods within the visualisation should be used to reinforce this hypothesis.

Animation is also used within Revision Towers to maintain context during changes within the display. The zooming operations available use animation to move smoothly between the start and end positions, and this is preferable to the discrete jumps that would otherwise exist. The other main use is to morph between the old and new layout when the current display becomes filled and towers must be resized. Animation here is also important. By morphing, the user is aware that towers have moved from their original positions, which may not be apparent if the new layout was presented directly – particularly if several towers were of similar appearance.

7.3.2. Evaluation using frameworks

Revision Towers will now be evaluated using the full framework of Storey et al. [Storey97] and the visualisation section of Knight's framework [Knight00a]. Each question will be answered using one of the following responses.

- *** - The feature is supported fully by the visualisation.
- ** - The feature is supported partially, with some significant aspects not included.
- * - There is minimal support for the feature, and such support is likely to be incidental.
- x – There is no support for this feature.
- N/A – The question does not apply to Revision Towers.

Evaluation of Revision Towers using Storey's software exploration framework produced the following results:

Element (Storey)		Corresponding Support	
E1	Indicate syntactic and semantic relationships.	***	Both relationships are indicated. Syntactic relationships are shown through the pairing and alignment of header and implementation files. Semantic relationships are shown by allowing the user to view files that were modified together.
E2	Reduce the effects of delocalised plans.	***	A change may affect a number of files. The visualisation supports highlighting all files that were changed at the same time. Files with the same comments or bug reference, or modified by the same author, may also be identified across the project.
E3	Provide abstraction mechanisms.	**	Details may be abstracted within a tower to cluster versions together. Towers could also be abstracted into groups, such as directories or modules, using a similar representation. However, this is not currently provided by Revision Towers.
E4	Support goal-directed, hypothesis-driven comprehension.	*	No annotation support for the tool has been included. The 'working' area provided by the visualisation does allow some hypothesis-driven comprehension, by allowing the user to examine possible files in further detail.
E5	Provide overviews at various levels of abstraction.	x	Very limited architectural information is available from the data source used by Revision Towers. Although directory and module views could be included using a similar representation, integrating Revision Towers with a tool such as 3dSoftVis [Riva98] may provide the structural information required.

E6	Provide views of multiple mental models.	***	A number of views are provided within Revision Towers. Within a tower, the author, file size and type of change are available. The timeline provides an alternative chronological view of the changes. Mouse over functionality provides low level details.
E7	Cross-reference multiple mental models.	***	The views are well cross-referenced. Selecting a release within a tower will show the corresponding release on the timeline, and vice versa. Towers may also be mapped to provide a more chronological viewpoint.
E8	Provide directional navigation.	**	The visualisation supports displaying all of the files in a single view. Navigation is therefore restricted to time, and the user is able to move through time in either direction at any speed.
E9	Provide arbitrary navigation.	**	As well as moving through time, the user may jump to a specific point in time using the timeline.
E10	Provide navigation between mental models	NA	As both the towers and the timeline may be displayed simultaneously, and both show the full extent of the project, there is no need to navigate between the two.
E11	Indicate the current focus.	*	The working area, and mapping towers onto the timeline, allows a maintainer to investigate some towers in further detail. However, the contextual information that may be required is not available directly.
E12	Display path that led to current focus.	x	No support for this exists. However, as every file may be viewed in a single display, the need for this is reduced.
E13	Indicate options for further exploration.	**	Further exploration is limited mainly to movement through time. The current position is shown on the timeline. Branches within towers are also highlighted, and these may be expanded as necessary.
E14	Reduce effort for user-interface adjustment.	***	Movement through time and zooming operations are animated smoothly, and new data included within the visualisation is shown without further layout changes. The layout algorithm used also reduces the need for a new layout should the data have evolved when the visualisation is next viewed.
E15	Provide effective presentation styles.	**	A Revision Tower is a clear and effective way of showing change log information. Filter operations also exist within the visualisation, although grouping is less well supported.

The application of this framework to Revision Towers has shown very mixed results. This is due partly to the fact that Revision Towers is not a traditional software exploration tool, and therefore some of the requirements are less significant. For example, E12 (display path leading to current focus) is required when arriving at a particular point in the system may be due to following a series of other links, such as control or data flow. As Revision Towers supports the display of all towers simultaneously, and does not have the same link structure, displaying the path is less important.

Other aspects may be more significant. The limited amount of abstraction available (E3, E5) has an impact on the size of the software that may be analysed using the visualisation. Providing support for allowing the user to create abstractions during the visualisation would result in a reduction of the complexity of the display, but due to the predetermined grid layout used, there would be no impact on the number of towers that could be shown. Removing the grid layout would remove this constraint, but instead decrease the consistency of the visualisation. The use of higher level overviews has been demonstrated in other visualisations, and so was not replicated within Revision Towers. Similarly, the annotation features required by E4 have also been implemented by other visualisations and would be suitable for use within Revision Towers with little modification.

Evaluation of Revision Towers using Knight's software visualisation framework produced the following results:

Element (Knight)		Corresponding Support	
1	Does the level of visual complexity reflect the visualisation metaphor being used?	***	Revision Towers uses a simple representation, with coloured rectangles used to represent the majority of the information. No unnecessary visual complexity is introduced. Transparency is used, but only to prevent other parts of the display from being obscured. Fading is also used, although this acts to reduce the effect of change blindness.
2	Is the visualisation able to scale to accommodate varying degrees of data?	**	Revision Towers is designed to show all of the data on a single screen. This has an impact on the total number of files that may be displayed simultaneously, with a realistic limit of around 200-400 files depending on screen size and resolution. Releases may be abstracted, so there is no limit to the number of releases that may be displayed. However, it is obviously not possible to display all of these simultaneously. The representation itself is best suited to files, as it is designed for header/implementation file comparisons. However, a similar representation could be applied to directories and modules.

3	In the first instance can the visualisation be generated automatically?	**	Automatic generation is crucial to a visualisation such as Revision Towers, where the data is changing on a regular basis. As the prototype tool demonstrates, the process of data acquisition and visualisation generation may be mostly automated. However, although an initial mapping between header and implementation file is generated automatically, human intervention may be required to refine this further.
4	Can the visualisation evolve in a meaningful way (i.e. within the constraints of the metaphor) as the underlying data changes?	**	The layout and representation used are specifically designed to handle evolution. Towers will grow upwards as new versions and releases are included. New files that are introduced to the project will always be shown after existing towers, resulting in a high level of consistency for those existing towers. Authors are also identified with the same colour throughout the lifetime of the visualisation. The visualisation does not currently support any changes to the data that occur after the visualisation has been provided with a data set. This would be necessary if the visualisation was to be run continuously.
5	Does the visualisation interface (underlying metaphor as well as implementation) facilitate easy interaction?	**	The representation used was designed to be intuitive, although no formal end-user studies were carried out. The visualisation is based on repetition of a simple structure, a revision tower, and so the entire visualisation may be understood once a single tower is understood. Animation has been used as a natural means of displaying the progression through time.
6	Is the representation used fully and completely documented in some way?	***	The representation is simple and relatively intuitive. Chapter 4 contains a full and complete description of the representation. Scenarios later in this chapter show how the resultant visualisations may be interpreted.

7	Is annotated information, over and above the graphics, available to the user of the visualisation in some way?	**	The user is able to access the underlying log information used to generate each section of a tower with a simple mouse over operation. However, further information, such as high level change information that may also be included with the project data, is not available.
8	Does the visualisation display extreme data (i.e. possible anomalies) with no problem?	**	Extreme data is displayed in an obvious manner, without affecting the display of expected data. Changes in size are handled by showing sizes relative to the file, rather than the project. This allows very large changes to be displayed without affecting other towers, although the tower with the extreme change will be impacted upon. The impact of this may be reduced through the use of a combination of absolute and relative sizes. Large or small numbers of versions may be accommodated using the existing table-lens like abstraction mechanism.
9	Can the visualisation be viewed as both an environment and as still views (even if the still views exist within the environment), under user direction?	***	Still views may be obtained by pausing the animation, or by generating screen shots. The nature of the animation means that the animation will only ever add information to the previous frame, rather than modifying or deleting content. This means that a still shot at any point in time will show a completely accurate image of the state of the software at that point, providing the time-based view is used. As blocks are faded in, rather than appearing suddenly, this also means that changes that will occur in the near future may also be visible from the static image.
10	Can the visualisation be viewed from more than one angle, at user discretion?	NA	This does not apply to Revision Towers.

Evaluation of Revision Towers under this generic visualisation framework provides more positive results, with some support for all of the required features. However, most of these features only have partial support within Revision Towers. The support for points 2 (scale), 4 (evolution), 5 (interface) and 8 (extreme data) may be improved by extending the current visualisation in the ways suggested.

These extensions may be completed independently, and would have no detrimental effects on other parts of the visualisation.

The remaining partially supported features, points 3 (automatic generation) and 7 (further information), are more difficult to provide complete support for. The data source used by Revision Towers to allow efficient generation means that very limited semantic information is available for automatic generation. Providing a fully automatic solution would require a far more extensive data set, with a detrimental effect on the efficiency of the generation. Similarly, adding further information requires additional demands of processing and human intervention in order to relate the new data sources to the existing log information.

Evaluation of Revision Towers against both frameworks together highlights a number of strengths and a small number of weaknesses. As a generic visualisation, it has some support for all of the required features indicated by Knight. Therefore, it is possible to assume that the well supported aspects of the software exploration framework, such as the displaying of relationships and delocalised plans, and exploration options across different views, will be well supported through the use of the visualisation.

7.4. HfVis Evaluation

This section will evaluate HfVis. Initially, an informal evaluation will be undertaken, examining a number of issues specific to HfVis that may not be covered by the other evaluation methods. Secondly, the visualisation will be evaluated using the frameworks mentioned previously.

7.4.1. Informal evaluation

As with Revision Towers, HfVis will be examined informally before the formal frameworks are applied. The same aspects will be addressed – those of the data source used, the effectiveness of the layout and representation, and the benefit gained from adding animation.

7.4.1.1. Data source

HfVis uses the data obtained from a source code fact extractor as the input to the visualisation. The visualisation was developed to show the changes arising within this data, over multiple releases of the software. However, it is important to evaluate how useful this data is, and whether alternative data would be more suitable.

Appropriateness of the data

Using the output of a fact extractor, rather than accessing the code from the version control repository directly, creates two initial problems. Firstly, each file within the repository will have multiple versions, and thus there will be a very large number of possible combinations of these files to make up a project. In order to limit the possible number of projects that may exist, the input to the extractor is restricted essentially to releases of the software. Therefore, changes occurring within single versions will be absorbed into one set of changes over the release. The effect of this is that data regarding

changes made at the same time to different files, and so indicating a probable relationship between those files, is no longer available. Secondly, a general issue with fact extractors is that pre-processor statements containing further configuration information are difficult to preserve. Within HfVis, some of this data may also be lost if definitions must be provided in order to allow the fact extractor to parse the code initially.

There is little that can be done to avoid the first issue, although having very frequent releases of the software will reduce the number of versions of a file contained between two releases. The second issue may be resolved by avoiding languages with a dependence on the pre-processor. However, as many major open source projects are developed in C or C++, this is not feasible.

The use of a fact extractor makes a significant difference to the visualisation, however. Instead of the lexical comparisons provided by Diff, HfVis offers a syntactic view of the differences between releases, and presents a number of advantages over a lexical view. In particular, using syntactic data allows the user to consider not only the number of lines changed in a file, but also the number of new methods in a class, or parameters that have been added to a function. This provides far more meaningful information than would be available by visualising the raw diff output alone. The effect of changes to the structure of the software may also be determined, which is very difficult to determine from traditional diff-based views. For projects with a low dependence on the pre-processor, the benefits of a more informative view of the changes within the release will more than make up for the problems involved in acquiring the data.

Additional data required

Although use of syntactic data is an improvement over lexical data, HfVis could be improved further through the use of semantic data. Currently, a function is only marked as changed in the visualisation if code within that function has changed. Adding semantic knowledge would alter this so that a function would only be marked as changed if the behaviour of the code within the function were modified. At this time, there are no tools available that would obtain this knowledge for real-world software.

However, program-slicing tools could be integrated into HfVis to provide similar benefit. As part of the query functionality, a slice of a particular variable or function could be requested. The representation would be modified so that current parts of the project affected by the slice would be highlighted, and the remainder dimmed. Playing the visualisation through time would then allow the user to see how the size of the slice has changed, in that nodes and bars within nodes would light up if they become affected at a later point. Additionally, changes would still be shown using the standard mechanism. Such behaviour may enhance the use of the tool for maintainers executing fault-finding tasks, by allowing them to compare specific sections of working and faulty software.

HfVis could also be improved by integrating the visualisation with a configuration management repository. The visualisation was designed to identify changes within files, and uses the source code to determine this. However, it may be more difficult to determine why those changes were necessary. By allowing access to relevant fault reports or mailing list archives, further information may be provided. Additionally, the type of data displayed by Revision Towers could also be integrated into HfVis. For

example, it is possible to determine whether files were changed together from the raw version control log. This information could then be used within HfVis to generate an alternative layout, where files that were changed together would be clustered. Such a layout may be beneficial prior to a reengineering process, as files that change together without a close calling structure may be candidates for restructuring. Similarly, the classification of changes made as determined from the comments field may be incorporated into a node, showing the type of maintenance undergone within the underlying file in the previous release. This could be shown with a series of small highlighted icons adjacent to the filename, representing the maintenance activities. Finally, this integration may allow a user to identify the author or authors of a specific function, which may be beneficial if the section of code was found to be deficient.

7.4.1.2. Representation

As with the rest of the visualisation, the representation has been designed to support evolution well, and be relatively intuitive. A node represents each file within the project, and each node has the same layout. Understanding a single node will therefore allow the user to understand the entire project. However, there are a number of issues regarding the representation that should be evaluated.

Absolute and relative sizes

As with the towers within Revision Towers, nodes are always of a fixed size. This simplifies the evolution process, as the use of a fixed size node reduces the need for future repositioning. In order to use this space most efficiently, and to ensure that changes within a small file are not overpowered by changes in a larger file, a combination of absolute and relative measures are used. The file size is shown as a percentage of the size of the largest file within the project, allowing both small and large files to be identified within the fixed size node. The file size bars are small compared to the size of the node, and so minimal amount of space is wasted for small files.

Within the node, five content bars are used to show the contents of the file for the current release, and the previous and future releases. These bars show each component within the file as a percentage of the makeup of the entire file, allowing each bar to be of a fixed height. This again simplifies the evolution process, as resizing of the bars to consider smaller or larger files is unnecessary. Additionally, changes are visible regardless of the size of the change, or the file. This is important as a single line change may have a large impact.

The combination of relative and absolute sizes allows the magnitude of each change to be determined visually, with efficient use of the available space. However, although suitable for a single node, it is more difficult to make cross-node comparisons than with the use of absolute sizes alone, for the same reasons as Revision Towers. Forcing content bars to be of a fixed height creates similar issues, in that two releases of the file may at first glance appear to be identical. The wider central bar reduces the problem for the release under consideration, as this allows absolute sizes to be shown also. However, this is not provided for the bars to either side, and the actual sizes for components within these bars must be determined using the file size bars. It is recognised that the need to examine two values in

order to identify the size of a component is less than intuitive, but comes as a result of the requirement to support viewing of multiple releases. If this requirement was removed, the fixed size constraints necessary for evolution could also be removed, allowing a more intuitive representation.

Use of colour

Colour is again an important factor within HfVis. The predominant use of colour is to show the makeup of the components within a file to ensure that a class is rendered differently from a preprocessor macro, for example. The colours may lighten or darken during the animation phase to show change within that component. These same colours, representing the same components, are used if the SeeSoft-like source code view is activated

However, colour is also used as part of the global metric display and also with the arcs used to link nodes together. As a number of metrics may be shown simultaneously, colour is required in order to uniquely identify each metric. Arcs are restricted to two colours, to show connections that exist and connections that were deleted recently. However, these colours may be the same as those used to show the file contents, leading to overloading and increasing the cognitive load on the user of the visualisation. Furthermore, the file view and class view also use different colours – red represents a class in the file view, and a private method in the class view. Therefore, switching between file and class views is also an expensive operation, and users should be advised of the change in meaning when the switch occurs. This will also have a negative impact on the intuitiveness of the visualisation.

7.4.1.3. Layout

Two different layout algorithms have been included as part of HfVis. A force directed placement graph is used to show an overview of the software, with arcs used to link all related nodes together. Such a view is appealing to developers, as the basic representation is one they are likely to be familiar with. Animating a force directed graph whilst it stabilises presents a fluid layout to the user. The constant updating of the graph allows nodes to be added and deleted without sudden changes to the graph layout, and so preserves context to some extent. The extent and direction that nodes move, however, is unrelated to the size or significance of the change.

However, the use of a graph results in issues of scale as the software becomes larger and more complex over time. Although the representation highlights recent changes to the graph structure, so ensuring that these are not missed, identifying other connections becomes progressively difficult. In order to reduce the difficulties of scale, further abstraction mechanisms should be introduced. These may occur manually, by allowing the user of the visualisation to group selected nodes together. Alternatively, the process may be automated through a process such as FADE [Quigley02]. Both of these cases would require small enhancements to the representation to show composite nodes. Grouped nodes could be shown with the content bars showing the composite content of the individual nodes. Changes to individual components within nodes would then be highlighted within this bar, allowing the user to drill down and investigate the change in detail. Similarly, an average size and metric value could be

used for the file size bars and metric views respectively. The change circle would be updated if any change occurred within the grouped node.

The purpose of the radial layout is to reduce the complexity of the force directed graph with large projects. By allowing the user to focus on a small number of related nodes, the user is able to investigate sections of the project in more detail. The layout is particularly suitable for evolution, as nodes moved to create space move in a predictable fashion around the rings of the layout, rather than in any available direction as is the case with a force directed graph. Therefore, the time for a user to determine the location of a moved node is likely to be reduced.

The main disadvantage of the radial layout is that it requires the graph layout to be forced into a tree structure. This is achieved by duplicating nodes where necessary, and therefore presents a much larger number of nodes than might be expected. These duplicated nodes are also animated in the same way as the original, although faded to reduce attention. However, the danger of this approach is that the user may still be attracted by a number of nodes undergoing large movement and so initially assume significant change in the software, when in reality many of these nodes are duplicates. As the initial assumption may be incorrect, the effectiveness of the visualisation is reduced. However, not animating duplicate nodes means the user may instead assume that no change occurred in the node, thus creating a similar problem. Therefore, an alternative view where nodes are not duplicated is also provided, although this introduces the need for the user to follow long, possibly crossing, arcs to view related nodes rather than these nodes being in close proximity. Neither view is therefore perfect for any situation. However, as the graph size increases through exploration, the benefits of node duplication are likely to outweigh the disadvantages.

7.4.1.4. Animation

Animation plays a significant role in HfVis, with two different uses. The first is as a means of maintaining context during layout changes that occur when nodes are added and deleted from the visualisation, when the user is viewing the evolution of the software. Zoom operations are also animated to maintain context.

The second use is more important. HfVis uses animation as the primary means of both representing and alerting users to changes occurring within the software. The file size and file content bars are both animated in such a way that movement will only occur when there is change. Additionally, the type of change that occurs, such as the inclusion of a new method in a class, is shown in a transient manner during the main animation phase of the visualisation. Problems of change blindness are removed by additionally marking any changes within the node so the user is aware of them, and providing the ability to replay the animation to allow the user to view the change again if necessary.

Removing the animation from the visualisation would have a significant effect. Firstly, unlike Revision Towers, only the release under consideration is displayed in detail, although some historical information is provided. Animation is used to change the release under consideration, by updating the nodes within the visualisation to reflect the current release. Therefore, without this, the user would be

restricted to analysis of a single release. A different release could then be selected, but the user would not be alerted to the changes that occurred as effectively as with animation. Instead, the user would need to spot changes by examining the entire display, which is a significantly more intensive operation.

Secondly, the representation would need to be modified to include the information regarding the type of change. Space within a node is limited, and therefore displaying the type of change within, or adjacent to the changed component would be difficult. Colour is already overloaded, and so further overloading would be unwise. The use of patterns would reduce the clarity of the display, and be difficult to identify accurately in the small space provided. Therefore, including the same level of information without resorting to a 3D display would require a very different representation.

By considering the effect of removing animation from the visualisation, it is possible to conclude that animation is an important aspect of HfVis. Without the animation, changes are difficult to identify and classify, and viewing a series of releases whilst maintaining context is very difficult. As the aim of the visualisation is to show changes over multiple releases, HfVis requires the additional expense of animation in order to achieve this.

7.4.2. Evaluation using frameworks

HfVis was also evaluated using the same frameworks as for Revision Towers previously. The same responses were available for each question, and are repeated below.

- *** - The feature is supported fully by the visualisation.
- ** - The feature is supported partially, with some significant aspects not included.
- * - There is minimal support for the feature, and such support is likely to be incidental.
- x – There is no support for this feature.
- N/A – The question does not apply to HfVis.

Evaluating the visualisation using Storey's software exploration framework produced the following results.

Element (Storey)		Corresponding Support	
E1	Indicate syntactic and semantic relationships.	**	Syntactic relationships may be displayed using the call graph or inheritance tree views. Semantic relationships are limited to viewing all of the files that underwent modification during the release, which may suggest that they may be related in some manner.

E2	Reduce the effects of delocalised plans.	**	The abstraction mechanism used, where header and implementation files are combined, may have some impact on reducing delocalised plans.
E3	Provide abstraction mechanisms.	*	Abstraction operations available for the user are limited to reallocating the weighting given to each component within a file. This allows increased display space to be allocated to more interesting components.
E4	Support goal-directed, hypothesis-driven comprehension.	x	This is not supported by HfVis.
E5	Provide overviews at various levels of abstraction.	***	The default display for each file shows contextual information, with the previous and future state of the file also shown. The level of detail implementation provides an overview of this data as the user zooms out. The global metric display also provides an overview of the file in relation to the project.
E6	Provide views of multiple mental models.	***	A number of views are provided within HfVis. Within a node, the user may select from viewing global metrics, a high level view of the changes within a file, a more detailed SeeSoft like view, or low level details. The layout additionally supports call graphs and inheritance trees.
E7	Cross-reference multiple mental models.	**	Limited cross referencing is available, as only one main view is provided. However, highlighting a node in the overview window will select that node in the main view, and vice versa.
E8	Provide directional navigation.	***	Within the radial graph view, the user may select a node to continue expanding the call graph with respect to that node. The user may also navigate through time in either direction at any speed.
E9	Provide arbitrary navigation.	**	Within the radial graph view, nodes not involved in the call graph are displayed as a distant outer ring. These nodes may be selected to allow a new graph to be viewed. The user may also navigate to a specific point in time using the timeline. No bookmarking support is included.

E10	Provide navigation between mental models	***	There are a number of views provided in HfVis. Individual nodes contain two of three selectable views of the data, to highlight short term and long term changes. The user may also change the layout algorithm used in order to gain a different perspective of the data.
E11	Indicate the current focus.	***	The node under consideration is centred within the radial or spiral views. Historical information is included in the node, showing how the file was modified to reach the current state. Additionally, with the radial view, related files are displayed in a ring around the current node.
E12	Display path that led to current focus.	**	Limited support for this exists. It may be possible to determine the path by viewing the radial graph and examining which nodes have been expanded. Within a node, historical data is shown along with the current data to provide increased context.
E13	Indicate options for further exploration.	***	Further exploration is available through selecting a node for investigation, expanding relationship information from a selected node, or selecting a different point in time.
E14	Reduce effort for user-interface adjustment.	***	Movement through time and zooming operations are animated smoothly, and new data included within the visualisation is shown without further layout changes. The layout algorithm used also reduces the need for a new layout should the data have evolved when the visualisation is next viewed.
E15	Provide effective presentation styles.	**	The node used within HfVis is a simple representation for highlighting changes occurring within a file at multiple levels of detail. The visualisation also supports grouping and filtering operations. However, there is an overloading issue regarding the use of colour.

The application of the software exploration framework to HfVis produces mainly positive results, with some support for most of the features required. This is due partly to the fact that HfVis is based on a tool much closer to that intended by the original framework, and so exploration and navigation tasks may be supported in a logical manner.

Two elements are lacking within the visualisation – E3 (support abstraction mechanisms) and E4 (Support hypothesis-driven comprehension). Issues regarding E3 with respect to scale were covered previously in section 7.4.1.3. Support for E4 could be added to the visualisation by providing annotation facilities, where the user may record additional comments to be attached to a node. Such a

facility would need to store the comments on a per-file and per-release basis, to indicate the change in behaviour of the file over the evolution of the software. Additionally, it would be necessary to preserve comments should the file become renamed.

Many of the elements with only partial support (E1, E2, E7, E9, E12, E15) could be supported better by integrating the visualisation into a dedicated comprehension tool such as SHriMP [Storey01] or Sniff+ [WindRiver03]. These tools are graph based, and provide many of the cross-referencing and exploration options currently lacking within HfVis. However, they currently have little support for evolution. As HfVis is also graph based, although with much greater use of a node within the representation, combining the two should be feasible. This would result in a more complete visualisation.

There would be a number of changes necessary to the original tool to support some of the existing tasks with respect to evolution. For example, the bookmarking aspect of E9 becomes a far more complex operation than within a static environment. Bookmarking a file would result in storing the name and release of that file. To be useful, activating the bookmark would then require the user to determine whether to view the named file within the current or original release. Should the file not yet exist within the current release, or have been deleted, the user would need to be alerted to this with further details. Similarly, should the file have been renamed, and another file given the original name, the user must decide whether to select the renamed file, or the file matching the original name of the bookmark.

Evaluation of HfVis using Knight's software visualisation framework produced the following results:

Element (Knight)		Corresponding Support	
1	Does the level of visual complexity reflect the visualisation metaphor being used?	***	HfVis also uses a simple representation, with no unnecessary visual complexity. Zoom operations and layout changes are animated, although this serves as an attempt to maintain context during these operations. Similarly, the process of viewing the evolution over time is also animated, but this again is done to maintain context.
2	Is the visualisation able to scale to accommodate varying degrees of data?	**	HfVis suffers from issues of scale that affect many other node and arc based representations. The radial graph view was designed to improve this, by showing only the nodes that a user is interested in. The use of zooming and level of detail support also increases the number of nodes available. Around 30 nodes may be shown in detail without scrolling.

3	In the first instance can the visualisation be generated automatically?	**	Automatic generation is crucial to a visualisation such as HfVis, where the data is changing on a regular basis. The use of tools that may already exist as part of the project reduces the probability that further human intervention, such as defining the pre-processor declarations, will be required. Once the data source exists, the visualisation is generated automatically, with no further intervention required.
4	Can the visualisation evolve in a meaningful way as the underlying data changes?	***	The layout and representation used are specifically designed to handle evolution. A node will change during the animation process to represent the current view, and nodes are of a fixed size to reduce possible impact on the display of other nodes. The layout will also evolve with new arcs and nodes inserted, and old ones removed as necessary. Finally, the fluid layout allows changes occurring at any point during the execution of the visualisation to be incorporated directly into the display – for example, if a new release of the software occurred.
5	Does the visualisation interface (underlying metaphor as well as implementation) facilitate easy interaction?	**	The representation used was designed to be intuitive, although no formal end-user studies were carried out. The visualisation is based on a familiar node and arc representation, although both nodes and arcs have been supplemented with further details relevant to viewing file changes.
6	Is the representation used fully and completely documented in some way?	***	The representation is simple, and relatively intuitive. Chapter 5 contains a full and complete description of the representation. Scenarios later in this chapter show how resultant visualisations may be interpreted.
7	Is annotated information, over and above the graphics, available to the user of the visualisation in some way?	**	The user is able to access the changes that were made at a textual source code level in order to obtain the full details. Separate annotation information is not available.

8	Does the visualisation display extreme data (i.e. possible anomalies) with no problem?	**	Extreme data is displayed in an obvious manner, without affecting the display of expected data. Changes in size are handled by showing sizes relative to that release within the file. An absolute view is provided simultaneously, where the size is shown proportional to the largest file size within the project. The combination of these two displays allows files of any size to be displayed and identified.
9	Can the visualisation be viewed as both an environment and as still views under user direction?	**	Still views may be obtained by pausing the animation, or generating screen shots. Some information is only accessible by viewing the animation, such as the number of new methods added to a class. Other changes are more easily identified when the animation is viewed. However, care has been taken to ensure that still views are not ambiguous. For example, although it is not possible to determine the exact change that occurred with the static view, it is possible to determine that there has been a change.
10	Can the visualisation be viewed from more than one angle, at user discretion?	NA	This does not apply to HfVis.

HfVis also has some support for all of the features required by the generic software visualisation framework presented. However, most of the features have only partial support within the visualisation. Many of these (points 3, 8 and 9) are related directly to the data and representation that was used, and are difficult to support fully without restricting the information that can then be shown. For example, full support for point 9 could be achieved by removing additional data provided during the animation. However, this would then reduce the effectiveness of the animation, and of the visualisation as a whole. Instead, the user must be aware of the limitations that exist when generating static images. Similarly, the benefits of increased accuracy and relevance provided by observing differences based on syntactic data require a small additional overhead of obtaining that data in a semi-automated process.

Integrating the visualisation with existing solutions as described earlier may support issues relating to points 2 (scale) and 7 (annotation) better. For example, additional abstraction options will have a significant impact on the number of files that may be viewed simultaneously.

Evaluation of HfVis against both frameworks highlights a range of issues. As a generic visualisation, it has some weaknesses related to the use of syntactic data, a graph-based display, and animation. These were used to improve the information content and evolution aspects of the visualisation. As a software

exploration tool, it also suffers from the emphasis given to the evolution aspects of the visualisation at the expense of some standard features.

However, solutions for the weaknesses identified exist within other research. Therefore, integration of HfVis with existing visualisations will resolve most of the weaknesses, whilst still supporting the new evolution features provided.

7.5. Feature-based evaluation

In addition to the formal framework-based evaluations of HfVis and Revision Towers, they were also compared against each other, and alongside other existing visualisations, in order to highlight the similarities and differences against existing research. The existing visualisations selected were chosen as ones that may be used to identify software change or evolution, and are described in detail in section 2.4.4.

A number of features were drawn out from the descriptions. Identification of important features was based on Maletic's software visualisation taxonomy [Maletic02], where the task, audience, data source, representation and medium are considered to be significant. Although the visualisations were selected as ones capable of displaying change or evolution, it is recognised that they may perform other functions also. However, these further properties have been excluded from this comparison.

The features identified, and the rationale for selecting these features, are described in the following table.

Feature	Explanation
Representation used within visualisation	A familiar representation, such as a flowchart, will be more intuitive to a user, reducing the required learning curve.
Number of dimensions required	A 2D system may be perceived as having a lower learning curve in terms of familiarity and navigation. 3D systems may require specialised displays and interfaces in order to be used effectively.
Is structured information indicated?	This is similar to element E1 above. By displaying some form of structure, such as a call-graph, changes may be viewed within the context of the software as a whole.
Main structure used within visualisation	This identifies the level at which the visualisation operates, and therefore if it is designed for overviews of the software, or detailed investigation.
Level at which change is identified	Although the actual change details may be available indirectly, this details the point at which a change makes a visual impression within the visualisation. This may allow patterns of change to be spotted at that particular level.

Type of change shown	This indicates the extent to which change may be identified within the visualisation. 'Exists' indicates that it is possible to determine that there was some change, but not what that change involved. 'Content' indicates that some property of the change within the main structure used can be identified. 'Structural' indicates that some effect of the change on the structure of the system can be identified.
Number of visible structures.	This feature indicates some idea of the size of software appropriate for the visualisation, and suggests the number of simultaneous comparisons that may be made. Displays are considered to be a fixed size, with no scrolling or zooming. A coarse scale of 1 < few < several < many is used.
Number of attributes displayed.	The ability to display a large number of attributes increases the information content of the visualisation, but also the complexity.
Number of simultaneous and total releases shown.	The number of simultaneous releases shown indicates whether the visualisation is useful for comparisons between two releases, or whether comparisons may be made with respect to several releases. The number of total releases indicates whether the user may also examine other releases in a consistent fashion within the visualisation.
Manner in which change is identified	Changes will be identified more quickly, and more successfully, if identified explicitly within the visualisation, rather than through a spot-the-difference exercise.

Table 7-1. Features to be identified in the feature analysis.

Table 7-2 shows the features contained within the systems mentioned in chapter 2. Diff and a CVS log file are also included for reference.

The ideal visualisation for change and evolution identification could possibly be based on a structured 2D graph, as this presents a familiar representation. Changes should be identifiable at every level, from modules to individual lines, and in full detail to provide maximum information content. Finally, the visualisation should also support many structures, and allow all releases to be compared simultaneously.

There are a number of conflicting features within this ideal visualisation however, and therefore emphasis must be given to specific tasks with the recognition that this may require compromise elsewhere. This is shown within the table, with each system having a different data set.

The strengths of Revision Towers and HfVis compared to other systems are evident from the table. Specifically, the heavy evolution focus given to these visualisations is apparent from the number of attributes and number of releases columns. The large number of attributes included within the display allows a range of changes to be identified, such as a change in author within Revision Towers, or a new

System name	Representation	Dimensions	Structured	Main structure used	Level that change is shown	type of change shown	#visible structures	#Attributes	#simultaneous, total releases
Beagle	Tree, Graph	2	Syntactic (Graph)	Files	Functions	structure	Many	1 (new/deleted release)	2,2
Evolution Matrix	Rectangles	2	No	Class	Class	content	Several	2 (file size, #vars)	all, all *
SeeSoft	Reduced Code	2	No	File	Line	content	Many	3 (length of line, level of indent, colour for metric)	2,2
SeeSys	Rectangles	2 + time	Physical (Treemap)	File	File	content	Many	3 (file size, 2 metrics)	1, all
SoftCities	City	3	No	File	Functions	content	Few	3 (file complexity, age, maintenance activity)	1,1
SoftVis (2d view)	Grid	2	No	Module	Versioned File	exists	Several	1 (release no)	all,all **
SoftVis (3d view)	Z-layered trees	3	Physical (Tree)	Module	Versioned File	exists	Several	1 (any-colour)	all,all
Software World	City	3	No	File	Methods	content	Few	Many (all available details of method)	1,1 ***
VRCS	Graph	3	Physical (Graph)	Versions, Releases	None	-	Few	3 (time, associated release, branching)	all, all
CVS Log	Text	1	No	File	Version in file	content	1	7 (author, time, comment, release, size, status, branching)	all,all
Diff	Text	1	No	File	Line	content	1	2 (change location, added/deleted)	3,3
Revision Towers	Revision Tower	2 + time	No	File	Version in File	content	Several	6 (author, time, maintenance type, release, size, branching)	all,all
HfVis	Graph, Percentage Bars	2 + time	Syntactic (Graph)	File	Functions, lines	content, structure	Few	4**** (file content, sizes, change type, age of change)	5, all

Notes: * restricted by display size. ** restricted by number of discernable colours. *** has some support for evolution. **** chart allows additional attributes to be shown

Table 7-2. Feature analysis of software visualisation systems.

global variable included within a file in HfVis. Combining this with the ability to view a large number of releases allows patterns of change of a particular attribute within the evolution of the software to be identified.

Unsurprisingly, these strengths cause weaknesses in other areas, when compared to other systems. For example, the Evolution Matrix and 3d SoftVis can both show changes across the system in a static 2D view, rather than requiring additional animation. Beagle, SeeSoft and SeeSys have an advantage over HfVis and Revision Towers in that they may show many structures on screen simultaneously, increasing information content and allowing more comparisons to be made. These systems also use simple representations that may be more accessible to end-users. Finally, Software World presents a very large amount of information about the contents of a file, so allowing many different types of change to be identified.

Revision Towers and HfVis are suitable for a range of tasks, as detailed in the following scenarios. However, as Table 7-2 shows, in some cases other tools may be more suited to a specific task.

7.6. Scenarios

A number of software engineering tasks related to change and evolution will be presented in order to demonstrate how Revision Towers and HfVis may be used in practice. These tasks are relevant to open source projects, and separated into two categories: a programmer working on the software, and a senior developer or manager with responsibility for the overall project.

Two real projects will be used to illustrate the behaviour of the visualisations. It is important to note that the purpose of presenting these scenarios is to demonstrate the behaviour of the visualisation, rather than analysing the projects themselves in detail. The projects used are:

A. Revision Towers. This was developed in order to demonstrate the Revision Towers visualisation. It is a small 10KLOC program written in C++, separated across 72 files and 45 classes. A single developer had access to the source code.

B. Allegro. This is a cross platform graphics library, available from http://alleg.sourceforge.net. It was developed by a single developer to be used on a single platform, but was then later extended to support many platforms by a large number of developers in an open-source environment. Written mainly in C, it is around 100KLOC in size, across 250 files in 12 directories.

7.6.1. Developer scenarios

7.6.1.1. Scenario 1

Scenario

The developer is working on an open source project for a period of time on a voluntary basis, and is very familiar with the structure and behaviour of the software. Other commitments mean they have to

leave the project for a short period. On their return, they need to be reacquainted with the changes made during the time, and be able to understand the effect of those changes. Having updated their mental model of the software, the developer is in a better position to make new changes successfully.

Revision Towers and HfVis provide the required information to determine the changes made, from two different perspectives.

Use of Revision Towers

In order to gain an overview of the changes, the developer should start Revision Towers with the complete CVS logs from the project. They should identify the point at which they left the project, either by playing through the animation or selecting a point on the timeline directly. An example of four towers, representing six files from project A, is shown in Figure 7-4 (a).

At this point, the developer should continue to play the animation until the end of the timeline. Figure 7-4 (b) displays the four towers at this point. The difference between these two pictures highlights the changes that occurred whilst the developer was absent from the project. Within the visualisation, fading would be used to highlight these changes, rather than the towers being displayed side-by-side.

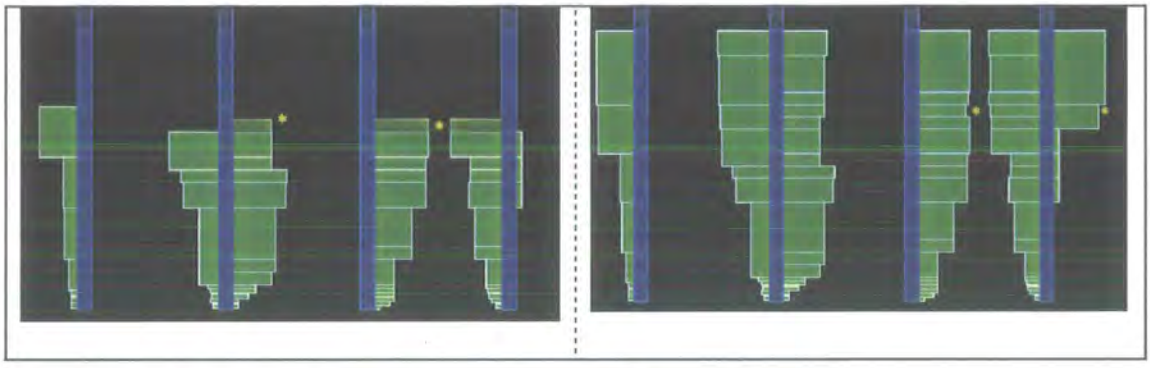


Figure 7-4 a) Prior to new changes made. *b) After new changes made.*

The files in green (from left to right) are animchar.h, animtype.h, animtype.cpp, recshape.cpp, shape.h, shape.cpp. (Additional annotations are in yellow)

In total, the developer was absent for four releases of the software, which may be determined by examining the distance from the lowest tower displayed to the top point of the tower. In addition, it is likely that a new version of the project has just been released, as there are no versions associated with the 'unreleased' section at the very top of each tower.

Examining the towers in more detail allows points about each to be identified. A small change has been made to *animchar.h*. Placing the cursor over the new section reveals that two lines were added, and one deleted. This change appears to have been successful, as no other changes were then necessary.

Much work has been undertaken within the *animtype* files. Firstly, the change that was just about to be made to the implementation file (starred, and shown as dark green within Figure 7-4 (a) as it is in the process of fading in) required no change to the header file, and so may have been fixing a problem from the previous release. A similar pattern emerged again after the following release. Therefore,

probably two fixes to this file were made in the developer's absence. Two small changes to the header file have also been made, together with the implementation file. This suggests that new functionality has been added to *animtype*. This functionality is likely to be small, given the small overall change to the implementation file.

Recshape.cpp has no header file. It is derived from the *Shape* class, and the class definition for *recshape* is also contained within *shape.h*. (It is not possible to access this knowledge from the log files alone, and so manual intervention would be required to connect *shape.h* and *recshape.cpp* together within Revision Towers). There is a high correlation of change between these two files, suggesting that most work has focused on adding new features. Since the developer's departure, a number of small changes were made over two releases, before the file stabilised. Of most interest to the developer may be the small temporary reduction in overall size of the file, which may warrant further investigation.

Finally, the towers representing *shape.h* and *shape.cpp* provide further useful information for the developer. The file *shape.cpp* has increased significantly in size over the time period. Furthermore, the main increase in *shape.cpp* (starred, shown as 27 lines when using the mouse over functionality) corresponds with the small decrease in *recshape.cpp*. (–8 lines). This suggests that some implementation may have been moved from the derived class *RecShape* to the parent class *Shape*. This could be determined more accurately by observing whether a similar pattern had occurred within the other derived classes. The header file has also continued to increase in size, with a small decrease also occuring. With the towers shown, no hints are provided as to why this may be the case.

The use of Revision Towers shows the overall picture of continuous development since the developer left the project. Much of this development has been implementing new functionality, although there is a suggestion that some restructuring has also taken place. In particular, the developer should take note of the large number of changes made to the parent Shape class, as these are likely to impact on many of the derived classes.

Use of HfVis

HfVis may also be used to show the changes in more detail. In order to use HfVis, the full source code from each release is required. The developer should start HfVis, and play through the animation until the point at which they left the project. In order to highlight the evolution most clearly, the radial graph view should be chosen. A section of this view, showing the same files as displayed in Figure 7-4 earlier, is shown as Figure 7-5 (a).

The developer should then continue to play the animation. Changes that occur will be shown using animation. Three further frames from this animation, each showing a particular release, are also shown as (b), (c) and (d).

A number of points may be determined by examining the frames above. (It should be noted that some of the changes, particularly within the file content bars, are identified more clearly when animated smoothly). Firstly, the graphs displayed within each node place the release within the context of the entire project, with the current release identified with a grey bar. A simple lines of code metric has

been used in the figures shown, with the red line showing the size of the header file, and the green line showing the size of the header file and associated implementation. These plots are both relative measures.

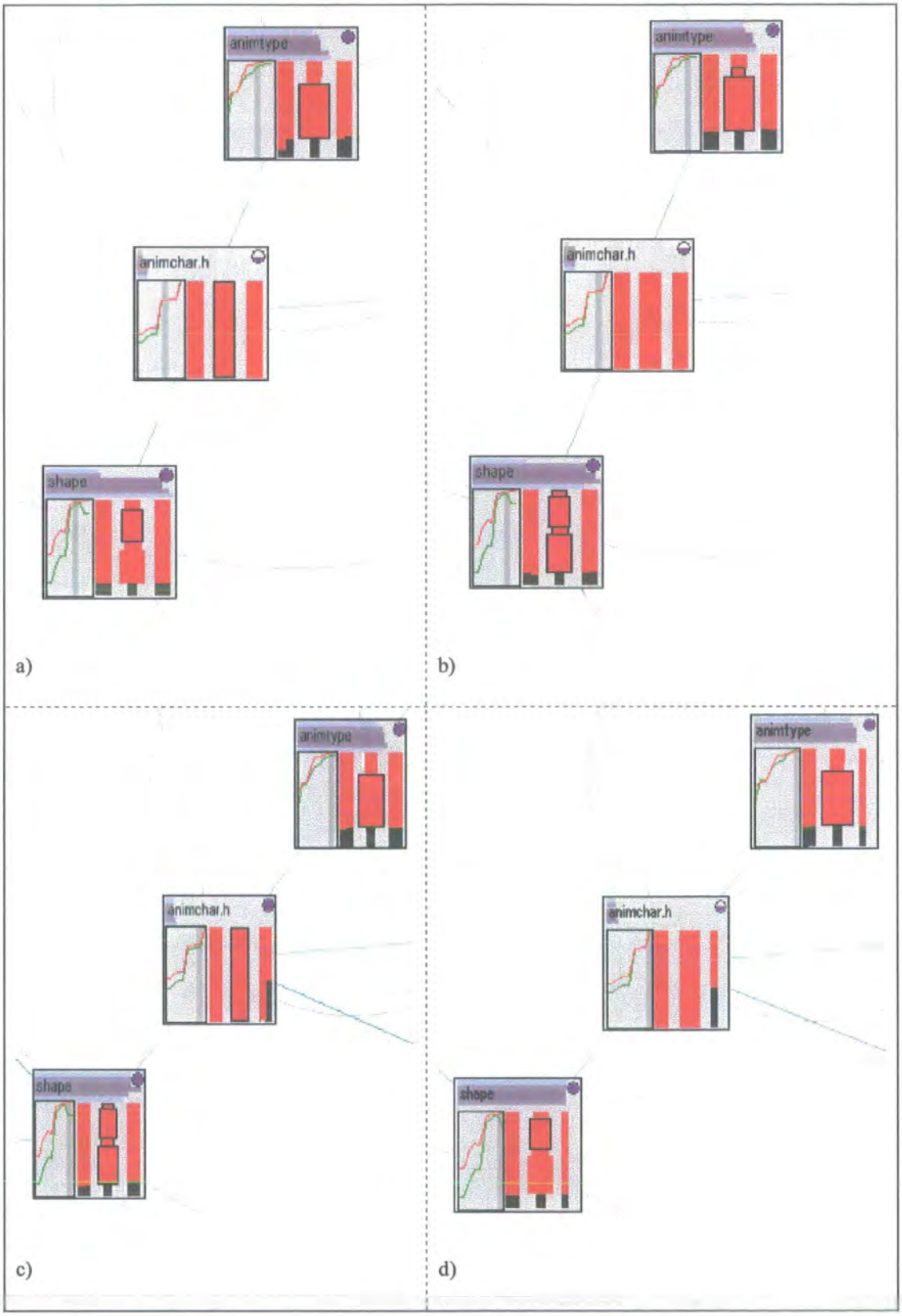


Figure 7-5. Four frames taken from the HfVis animation of animtype, animchar.h and shape over four releases.

Therefore, the developer may determine from the metric graph that, relatively, most of the implementation has taken place within the *shape* and *animchar.h* files since their original departure from the project. Analysis of *animchar.h* in further detail shows that the file continues to contain just one class for the majority of the time, as shown by the large red block. However, Figure 7-5 (c) shows a number of #defines will shortly be introduced to the file – indicated by the grey block at the base of the rightmost bar in the node. The size of the class has continued to grow, but very slowly. This may be determined by analysis of the graph, and of the purple file size bars shown at the top of the node.

In addition, the darker blue line in Figure 7-5 (c) leading out from the node shows that *animchar.h* now includes a new file. As these lines lighten over time to allow emphasis to be given to new changes, it is possible to determine that this is the first structural change to *animchar.h* for a number of releases. The file *filter.h* (not shown) was also a new inclusion for the *shape* node – also shown with a blue line. All of the classes within the shape were also changed, although it can not be determined from this view whether the changes are as a result of the inclusion. However, given the impact of adding *filter.h*, it may be worth the developer investigating the nature of this file in more detail.

Animtype has also continued to grow slowly, as expected from comparisons with Revision Towers. The structure of the file has remained relatively constant, although a number of new #defines are continuing to be introduced. Although shown better through observing the animation, it may be seen in the static frames as a slight grey incline at the base of the file content bars. Additionally, as the name *Animtype* is shown without a suffix, it indicates that the node also contains the implementation for any part of animtype, regardless of the physical file in which it appears.

Finally, the *shape* node has undergone few structural changes, although it has grown significantly in size. The *shape* node is particularly complex, containing four separate classes and a small number of #defines. In particular, the new class introduced within the file shown in Figure 7-5 (a) grew a little in size – both in terms of number of methods, and lines of code. Similarly, the fourth class also grew. This may be determined by the slight reduction in height of the first and third classes, suggesting that the second and fourth now take up a greater percentage of the file as a whole. In addition, the possible change identified within Revision Towers of implementation moving from *recshape* to the parent *shape* class appears not to be the case. This may be determined by a small reduction in size in the *recshape* class within the *shape* node that was not mirrored in the other containing classes.

7.6.1.2. Scenario 2

Scenario

The project has been in a state of evolution for some time. As the software continues to grow in popularity, new features are requested that were unconsidered in the original requirements and design. Adding these new features has changed large sections of the code in unpredictable ways. The reliability of the code has since decreased, and therefore the developer wishes to identify these areas undergoing significant change in order to focus their efforts on problem areas within the software during a period of preventative maintenance.

Use of HfVis

HfVis is well suited to providing some answers to this problem. Figure 7-6 highlights the animation of a single file taken from project A.

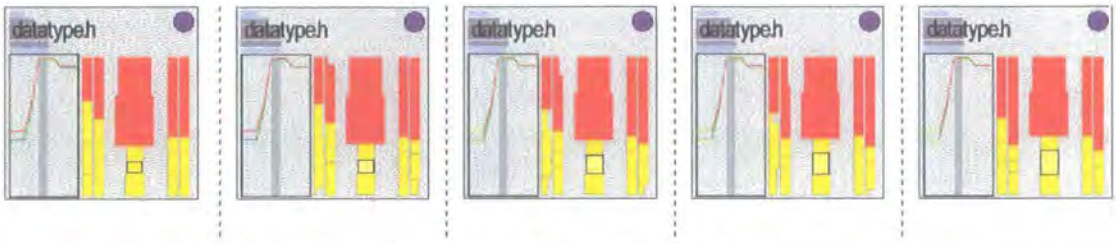


Figure 7-6. Five frames of the animation of datatype.h over a single release.

This figure highlights a number of issues within the file that may be problematic. The file, *datatype.h*, was originally designed to define a small number of structs used elsewhere within the software. These structs were simple – for example, the *Coord* struct required two integers to act as a coordinate. An initial glance of the figure above shows that, as well as the narrow yellow column at the base of the middle bar showing the small structs, two larger classes now also exist within the file. The second of these is of a significant size, and should almost certainly be moved to a separate file. Therefore, in general, the file is more complex than it was originally designed to be.

Closer examination of the node under consideration reveals further information. The central bar represents the change between the current and the next release. The bars to the left show previous changes, and the bars to the right show future changes. Therefore, it can be seen that two releases prior to the one under consideration, one struct was removed (at the top), and one introduced (at the bottom). In the previous release, a new class of significant size and number of methods was added – noticeable from the width and height of the equivalent section in the central bar.

Within the central bar, the second struct changes significantly in height, but very little in width. This would indicate that a number of new attributes have been added to the struct. Further examination reveals that this is the *Coord* struct, and therefore the developer should investigate why these further attributes were required for a simple structure.

The next release to the one under consideration did not change, with the graph to the left showing a straight line at that point. Finally, the last release shown shows that the bottom struct has been removed from the file. This struct is the same as the one introduced in the first release, and indicates two possibilities. Either the functionality was introduced, and found to be problematic or unused, or that the struct was moved to a different file. Using the query operations provided, the developer may determine that the struct has been moved elsewhere. However, more significant is the fact that it was added and then moved, and so would probably have been better being added to a new file in the first place.

The high level of change associated with this file, and the increase in complexity of many of the components within it, means that it should be recognised as a serious candidate for restructuring.

Use of Revision Towers

For comparison, the Revision Towers representation of the same file is shown as Figure 7-7.

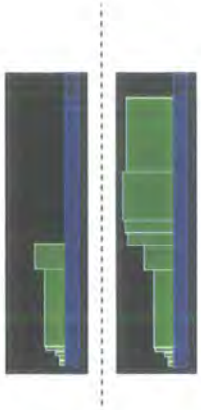


Figure 7-7. Two frames of the animation of datatype.h in Revision Towers.

The picture here is much less informative than that presented using HfVis. The two versions of the file made in a single release, shortly after the first frame, may raise some concern about the regular changes made to a stand alone header file. Also, the decrease in the width of the tower in the latest release of the file also indicates that some code may have been removed.

This final reduction, particularly when a separate implementation file does not exist, would indicate that the header file may contain unnecessary structs or global variables that were removed. The developer may therefore decide that further investigation into the file is required as to why this deletion occurred.

However, whereas in HfVis the significant structural and behavioural changes to the file were obvious, these are hidden when using Revision Towers. Furthermore, without the final reduction in size, there would be little to suggest that the file was a good candidate for preventative maintenance.

This scenario therefore highlights the differences in the data used by Revision Towers and HfVis. In this case, HfVis provides the most informative information. However, Revision Towers may be useful in other cases. For example, the regular changing of *shape.h* without a corresponding change in *shape.c* in Figure 7-4 (b) would indicate that restructuring of this file might be required. Therefore, in order for the developer to identify suitable candidates for restructuring, information from both HfVis and Revision Towers should be used.

7.6.2. Manager scenarios

7.6.2.1. Scenario 3

Scenario

Due to the departure of a developer acting as project manager in an open source project, a new developer has taken over the role of project manager. Although they are familiar with some parts of the project, they now require a greater awareness of the software as a whole.

Use of Revision Towers

The manager should use Revision Towers to generate an image of the project. The high level view, and the ability to see the complete history of the project, allows a much better overview than would be possible using HfVis. Figure 7-8 shows the complete image of project A. Additional annotations have been shown in yellow.

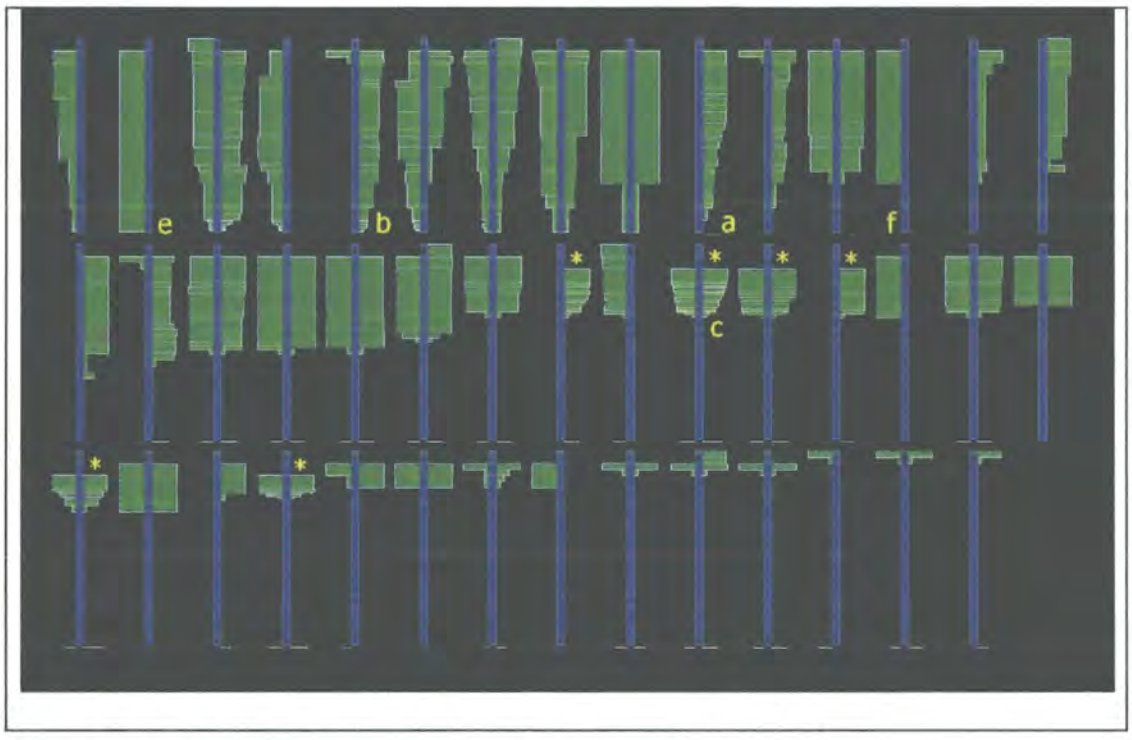


Figure 7-8. Final frame from Revision Towers, visualising the whole of a project

Using this picture, the following questions may be answered.

1. How many files are in the project?

72 files are in the project, determined by counting the number of towers.

2. How many authors are involved?

Only one author is involved, as every version in every tower is the same colour (green).

3. Which files undergo the greatest change, in terms of versions and lines of code?

In terms of the number of versions, files *a* and *b* – identified by the number of sections in each tower. Additionally, these are also some of the oldest files in the project. In terms of lines of code, files *a*, *b* and *c* have all increased significantly in size. The triangular shape of the towers indicates sustained growth, and the longer light lines at the bottom of the towers indicate a high scaling factor when drawing the tower.

4. Which areas of the project are most active?

The files represented by the towers at the bottom right of the image are the most active. Although only recently added to the project, many of these towers have very recent modifications that have not yet been released – indicated by sections at the very top of the tower. Many older files, and file *b* in particular, are also active with many recent modifications. Prior to the last release, many of the files marked by a star were very active – indicated by the small size of many of the sections within the tower.

5. Which areas of the project are least active?

The towers marked by a star also indicate that these files were not associated with the last release – shown by the lower height of these towers compared to the others in the project. Examination of the filenames shows a common prefix, suggesting the files were part of the same module. The fact that they are no longer associated with a release suggests that the files have been separated off into a new project.

There are also a number of files that have undergone very little activity. Files *e* and *f* in particular have not changed, but are still associated with the project. This lack of change, compared to the other files of a similar age, suggests that the files may not be associated with the project, and should be removed. This would need to be determined by further examination of the role of the files.

7.6.2.2. Scenario 4

Scenario

The open-source project has been in development for some time, and has increased in popularity. This popularity has increased awareness of the project, and a number of new developers have contributed patches to repair and enhance the behaviour. A senior developer, acting as a project manager, wishes to investigate the roles of the new developers within the project, in order to identify whether improvements can be made.

Use of Revision Towers

The majority of the data useful for this scenario is again held within the configuration management system, and therefore the manager should use Revision Towers. A small number of releases, from part of project B, are used as an example. Figure 7-9 shows a number of frames of the animation of these releases.

This sequence highlights some interesting behaviour within the project. Colour is used to identify the author associated with the check-in of a particular version. Within Figure 7-9 (a), all of the files have been checked in most recently by the yellow author. In Figure 7-9 (b), the light red author has checked in many of the files, with the yellow author making no contribution. Figure 7-9 (c) shows the cyan author made most changes, and Figure 7-9 (d) shows the dark red and green authors made many changes. Finally, Figure 7-9 (e) and 7-9 (f) introduce a further author shown as light green, although they also include changes from other authors, such as the yellow author appearing in the second last tower in 7-9 (e).

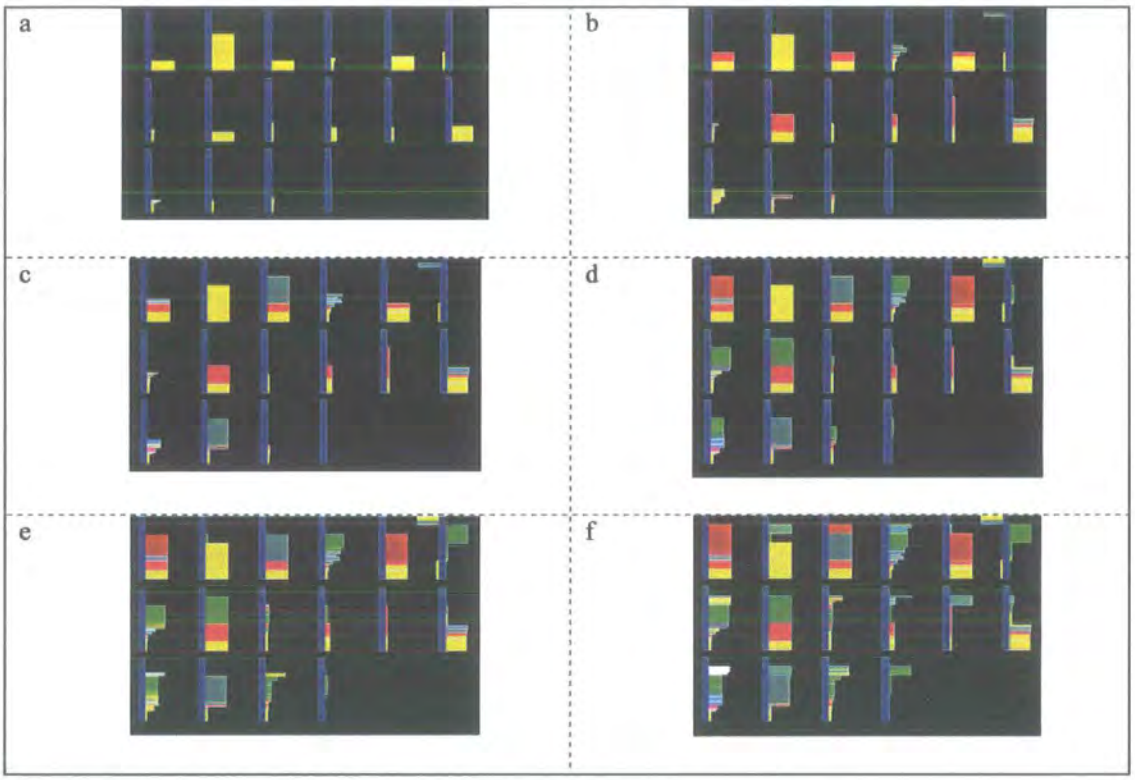


Figure 7-9. Six frames taken from Revision Towers, visualising part of project B.

The visualisation firstly indicates that there is no fixed ownership of the files. Rather than a single developer being responsible for the maintenance of a file, developers may submit patches to any files. With a large number of developers active in a single file, this means that there may not be a single developer that is fully aware of the current state of that file. Currently, only a small number of developers are involved within each file and therefore this is not a critical problem. However, as the project grows, it may become necessary to move to a project management style where check-in rights to a file are possessed by a single developer, who is then fully aware of all changes being made to that file.

Secondly, the sequence indicates that a single developer was often responsible for many of the changes made during a release. This suggests either that the overall ownership of the project was changed on a release by release basis, or that a change made by a developer affected a large number of files. Common sense would suggest that the first of these options is very unlikely. Therefore, it may be assumed that the changes made were far reaching, affecting a substantial part of the project. This

assumption may be confirmed through use of the query functionality provided – in particular, by examining whether the versions in different files had the same comment and timestamp, indicating that they were checked in together.

Assuming these changes were successful, this may indicate developers with a substantial knowledge of the project capable of making far-reaching modifications, and so may be suitable for a module ownership role. Alternatively, it may be that the change made could have been localised more to reduce the effect on the project, and future changes by the developer should be treated with caution. Although Revision Towers can not provide the answers to these questions, it allows the manager to be aware of the situation. Additionally, integration of the visualisation with more detailed file comparison tools allows the manager to investigate further.

7.7. Summary

This chapter has provided an evaluation of the visualisations detailed in chapters 4 and 5. The ideas were evaluated using a number of techniques in order to examine as many features of the visualisations as possible.

Firstly, the visualisations were critically evaluated on an informal basis. This critical evaluation highlighted the difficulty of providing high information content in the visualisation whilst maintaining a display with low visual complexity. The use of animation was also justified, with significant benefits of adding animation to both visualisations.

Secondly, the visualisations were evaluated using existing evaluation frameworks. Although neither framework was designed for animated, evolutionary visualisations, both frameworks were sufficiently flexible to be used for this purpose. The frameworks highlighted that although Revision Towers and HfVis are suitable as stand alone visualisations, they will be more effective when integrated with other tools.

Thirdly, the visualisations were compared to existing systems that could also be used to show aspects of change and evolution in software. A number of features significant to visualisation, change and evolution were identified that would be part of a 'perfect' visualisation. The result of the analysis of these features highlights not only the difference between HfVis and Revision Towers, but also how they complement existing systems. The emphasis given to the use of evolutionary, rather than static, data is also clear from this comparison.

Finally, four scenarios were presented demonstrating the use of Revision Towers and HfVis in practice. These scenarios show the information that can be obtained from the visualisations, and the type of hypotheses that can be made. However, they also demonstrate the information that is not available directly, and so the need for additional tools in order to investigate some areas of the project in more detail.

Overall, the evaluation has shown that both visualisations were successful in visualising aspects of the evolution of software, and using animation in order to achieve this.

ANIMATING THE EVOLUTION OF SOFTWARE

8. Conclusions

8.1. Introduction

The aim of this research was to investigate and develop visualisations targeted towards evolving software. The visualisation of evolving software raises a number of new issues relating to the difficulties of dealing with continually expanding data sets. Therefore, a major component of this thesis is a discussion of the methods and techniques that may be used in order to solve some of these problems. The problem of continuous, unpredictable data expansion is a difficult one, and there are perhaps many solutions. However, it is the development of two of these solutions that represents the remainder of the work in this thesis.

The developed visualisations are two dimensional, and use animation to show the progression of time and the effect on the project at each release. The use of animation within visualisation to show any form of changing data also introduces a number of new issues. Perhaps most significant is the concept of change blindness. This is the principle that differences between two displays may not be identified unless the user of the visualisation is focusing upon the changed area at the time of change. This has a significant impact upon animation, where it is not possible to assume that the user will see every change as it occurs. Representations must therefore be designed to accommodate this concept.

There are a number of possible extensions to these visualisations, such as introducing 3D displays or using evolutionary data based on dynamic, rather than static, properties of the software. However, these were not considered in this research. Instead, the focus was given to the layout and representation issues involved when developing evolutionary visualisations, with the aim of allowing the process and effects of evolution within a software project to be better understood.

8.2. Research Summary

This research has presented a number of issues related to visualising evolving software. The primary contributions may be considered to be:

- An investigation of the key differences of visualising evolving software, as opposed to visualising a single release.
- A demonstration of the feasibility of the use of animation to highlight evolution, particularly when the size and extent of the evolution is unknown.
- The description and evaluation of two visualisations that allow the evolution of software to be viewed at two different levels of granularity.

The two visualisations, Revision Towers and HfVis, both use two-dimensional representations combined with animation in order to show the evolution of software. However, although both visualisations have the same overall aim, some of the techniques used to achieve these aims are very different. These techniques are based on those identified as important to evolutionary visualisations, as described in chapter 3.

Revision Towers uses typical log data available from a configuration management system as the primary data source. A tower structure is used to display version information relating to files contained within the system. Each tower has two sides, allowing the details of header and implementation files to be compared alongside each other.

The height of a tower is fixed, and the vertical space is then allocated to the releases that have occurred during the lifetime of the project. Each side of the tower then contains a series of sections, and shows how the different versions of the files map to the releases of the project. The width of each section is mapped to the size of the version, and so shows how the file has grown or shrunk over time. Finally, each section is coloured according to the author responsible for the version.

A big-box approach to managing evolution is used. The individual towers are laid out in a grid formation in chronological order. Space is then left unallocated at the end of the grid to allow for future files to be added without disrupting the existing layout. As each tower is of a fixed size, abstraction techniques are included in order to maximise the use of space within each tower. Finally, animation is used to fade in parts of the tower at an appropriate time within the visualisation. This allows the development of the project to be viewed. As demonstrated in chapter 7, the visualisation is useful for viewing evolutionary patterns within the project, over a period of time.

HfVis uses the data taken from a fact extractor applied to the source code of a number of releases. A node-and-arc representation is used to display the data, with a node representing a file and arcs used to show the call structure.

The content of the node displays a number of details of the file. The current file size and an indication of the file content are shown over a period of five releases, allowing any change occurring over a short period of time to be observed. Each node also contains a chart displaying metrics for the complete history of the file. This chart may instead be replaced with a SeeSoft-like view of the source code of the file for the current release, if required by the user.

Changes are identified on a syntactic, rather than lexical basis. These changes are shown both in the context of the source code of the file, and also the structural effect that they caused. Animation is used to highlight the changes between a release and the previous and subsequent releases, by allowing the user to play through the historical data. Animation is also used for necessary layout changes as new files are introduced, and old ones deleted. An animated approach to managing the evolution is therefore used, although abstraction is also applied in order to reduce the amount of animation required. As demonstrated, this visualisation is more useful for locating changes precisely, and viewing structural changes that have occurred as a result of these changes.

Both visualisations were evaluated using a number of techniques. As well as an informal evaluation, the visualisations were evaluated using existing software visualisation frameworks. The visualisations were also compared against other existing visualisations focusing on software evolution. Finally, a number of scenarios were presented, showing how the visualisations may be used to gain a better understanding of a project.

Although the two visualisations are distinct, there are a number of important common elements within them, which will apply to other evolutionary visualisations where animation is used to show the progression of time.

The importance of consistency

Consistency is critical within these visualisations. When playing through the animation, it is vital that the layout remains as similar as possible, in order for the user to view the evolution in context. Large layout changes will create confusion, as the user becomes lost due to the relocation of familiar reference points.

The concept of comparison blindness, making identification of the differences between two separate displays difficult, means that consistency is also important when the visualisation is executed using an extended data set. 'Future-proof' refers to the need for a visualisation to appear similar each time it is executed. As a user becomes familiar with a visualisation, they will learn the location of specific elements, and the meaning of particular colours and patterns. It is then inappropriate to change these locations and colours simply because the data set has been enlarged, as this would mean that the user would have to reject their previous understanding of the visualisation completely. They would also then have to reacquire the same knowledge for the part of the extended data set that they had already been familiar with.

Fixed size elements

Fixed size elements are a key component of the representation for both Revision Towers and HfVis. Revision Towers uses a constant sized tower, and scales the versions and releases appropriately in order to display the data. HfVis uses a fixed size node, and uses percentage bars to show the actual file content. These percentage bars are also guaranteed to be of a fixed size, and so allow all the information known, or unknown, to be displayed without requiring further resizing.

The benefit of fixed size elements is that the impact on the layout when animating the evolution over time is significantly reduced. Additionally, the limited effect on the layout also means that the layout aspect of future proofing may be achieved more easily, as new data should not require more space than the previous data.

The disadvantage of fixed size elements is the effect of extreme values that must be mapped to the fixed size. If the ratio of the data value to the size of the space is made on a project wide basis, then a single large file will dominate the visualisation. If, instead, the ratio is made on a file-by-file basis, then comparison of values between files is difficult. The solution used within both HfVis and Revision Towers is to show both a file-by-file ratio and a project-wide ratio simultaneously as part of the representation used. This then maximises the space available, whilst still allowing file-by-file comparisons. Although this representation still has some problems, the reduction in layout changes makes it a more feasible solution than variable sized elements.

Static images and change blindness

As both Revision Towers and HfVis demonstrate, animation is a viable way of showing the evolution of software over time. Nonetheless, it is important to recognise that a paused animation must be informative. The motivation for this is that the developer may view the animation to gain an overview of the modifications made within the project, and identify possible trends and patterns. However, it is not always possible to confirm these patterns unless the animation may be stopped at a particular point. For example, without this ability a mouse over operation on a node may become difficult, as the node is continually moving.

Alternatively, the focus of the user on one aspect of the software, particularly during zooming operations, will mean that animations elsewhere may go completely unnoticed. The user must therefore be aware of these changes, even if they did not witness the change as part of the animation process – perhaps due to change blindness. This awareness may be provided by the provision of static elements within the animated representation.

The need for static images will have an impact on the representation used. If a data attribute is mapped only to animation, such as oscillation, there must be a means of accessing this data attribute statically, for example, by the use of mouse over operations. Similarly, if animation is used to show how the data has changed – for example, that a number of lines of code have been introduced or deleted, then it is necessary to show that a change has been made. In this case, if the number of new and deleted lines was the same, then the final image may appear the same as the initial one. Therefore, it is necessary to add an additional element to the visualisation to show that some change has occurred, even if the user did not witness it.

8.3. Criteria for Success

In light of the visualisations developed, and the common elements that may apply to other evolutionary visualisations, it is possible to review the introductory chapter in order to summarise what has been achieved. Several criteria for success were presented in section 1.3. They will now be re-examined in order to demonstrate the extent to which they have been achieved within this thesis.

a) Identification of the benefits of visualising evolving software.

This research has identified two main benefits of visualising evolving software. The first is that visualising evolving software can act as a means of identifying software change, by providing the ability to compare several versions of the same file. As demonstrated in section 2.3.3, there are a number of techniques that exist for identifying differences, with the aim of reducing the difficulty of software maintenance for a developer. Therefore, visualisation of evolving software can also ease the maintenance task.

The second is that the process of evolution, both within a specific software project and in general, may be better understood. As section 2.2 highlighted, there are a number of features that may be identified in existing software that can indicate that remedial action is required. However, these features are not

always easy to identify on a day-to-day basis. However, taking a step back from the software and viewing the evolution over a period of time may identify the trends more clearly. For example, an intensive period of development may introduce a large number of faults. Visualisation may be used to highlight this intense development, and so warn the manager of the need to plan fault fixes.

b) Identification of the key aspects when visualising evolving software.

A number of key aspects for developing evolutionary visualisations have been presented in chapter 3. In particular, the concept of a 'future-proof' visualisation has been introduced. This recognises the inevitability of the software evolving, and therefore the requirement for a visualisation to also support this evolution by adding the new data without causing significant disruption to the existing layout. This concept has far-reaching effects, and is the critical difference between static and evolutionary visualisations. Specifically, section 3.4 demonstrates how providing 'future-proof' support in a visualisation may have a big impact when deciding upon a suitable layout and representation.

A further aspect identified is the difficulty of supporting a real-world metaphor completely within evolutionary software visualisations. As software is virtual, it may evolve in ways that are impossible to map onto a physical environment. Therefore, the close mapping required for a metaphor between the behaviour of the representation in the visualisation, and the behaviour of the underlying data, is lost.

c) Assess the suitability of animation within software visualisation.

The use of animation within software visualisation has been restricted mainly to algorithm animation as detailed in section 2.5.1 and such animations have been shown to be successful. Animation has also been shown to be useful as a means of representing attributes of data within information visualisation – for example, by mapping a value onto the frequency of oscillation of the element within the visualisation. Animation has been used for maintaining context during zooming or layout changes, described in section 2.5.2.

This research has applied many of these techniques to software evolution. In particular, animation has been identified as one method for achieving the 'future proof' property recognised. Animation has also been used as a natural means of showing progress over time, by using animation to show software evolution. A number of issues that must be considered when using animation have also been identified in section 3.6. The use of animation within the two developed visualisations (sections 4.4 and 5.4) demonstrates the feasibility of the approach.

d) Development of new visualisations highlighting software evolution.

Two new visualisations have been developed within this thesis. These visualisations support the investigation of several releases simultaneously, allowing modifications to be identified over a period of time. The implementation of proof of concept tools for these visualisations is detailed in chapter 6, and the use of the visualisations in a number of scenarios is described in section 7.6.

Revision Towers (chapter 4) provides a high level view of evolution, based on configuration management data. This view allows the evolution process to be observed across the project, and allows general trends to be identified.

HfVis (chapter 5) provides a low-level view of evolution, based on syntactical differences within the source code. This view allows changes to be identified within the context of an individual file or class, as well as within the context of the rest of the project.

e) Address issues of scalability to be suitable for real world projects.

Consideration has been given to the representation and layout of both visualisations in order to make them appropriate visualisations for both small and large projects. In particular, the size of the project may vary considerably as it evolves – both in terms of number of files or modules, and in the number of versions and releases that exist, and the visualisations recognise and support this. Revision Towers uses a number of abstraction techniques in order to maximise the information content with respect to the size of the project. HfVis uses similar techniques, and also introduces a contextual radial layout to allow the user to focus on specific areas of the software whilst continuing to support evolution.

f) Assess the feasibility of automatic generation of the visualisations.

Chapter 6 provides implementation details of proof of concept tools for both visualisations. These tools illustrate that obtaining the required data requires little or no human input. The tools may then produce visualisations that are tailored to the size of the data received and that are consistent with previous versions of the data set. In addition, the use of animation in both visualisations is fully supported within the tools.

Re-examination of these criteria with reference to the appropriate parts of the thesis has shown that this research has been successful. Namely, that it addresses the overall aim of developing and evaluating visualisations that allow the differences between releases of continuously evolving software to be identified, and adds to the current software visualisation research.

8.4. Future Work

Although this research is self-contained, there are many further directions available for study. Reference has been made to many of these elsewhere in this thesis, but they will be examined in this section in further detail.

1. Results from use of the visualisations

This thesis concentrates on the development of visualisations to show evolving software. The next logical step is to apply these visualisations to a wide range of existing projects in order to identify evolutionary trends.

A number of studies may be possible. For example, the visualisations could be used to examine the effect of refactoring sections of the code, in order to implement various design patterns. The continued evolution may then be studied visually, allowing further analysis of whether the evolvability of the project as a whole increases, or whether the introduction of the design pattern causes other problems elsewhere in the software.

Similarly, studies could be carried out into the effect of employing different programming languages. Revision Towers is limited to use with a language with header and implementation files. HfVis is targeted more towards C and C++, although similar representations could be used for other languages. Again, this would allow research into whether particular languages have fewer problems of evolvability, in terms of the time to complete modifications, and the number of faults generated from these.

Finally, the visualisations may be used to show general differences between closed and open source development. At this time, open source data does not have enough history in order to determine the extent to which the existing laws of software evolution apply. However, as projects reach the end of their life, it may be possible to then study these systems more accurately, in order to determine the effectiveness or otherwise of using an open source development model.

2. Evaluation frameworks

There has been little research into frameworks that are suitable for evaluating software visualisations. Furthermore, there has been even less into the two main issues within this thesis – those of evolution and animation. A number of evolutionary visualisations now exist. However, without support from evaluation frameworks, or better, empirical evidence, these visualisations are unlikely to be accepted. Therefore, it is a pressing concern that new frameworks, targeted specifically towards evolutionary visualisations, are designed and evaluated before more research continues in this area.

3. Further evolutionary visualisations

This research has examined the use of animation to develop two evolutionary visualisations. However, there are more data sources that may be visualised that would provide more detailed information as to the nature of software evolution. For example, the combination of dynamic data integrated into an evolutionary visualisation would allow the changes in control or data flow to be viewed when the program was executed. This may allow easier identification of faults that have been introduced to previously working areas of the software as a result of unexpected impacts. Alternatively, a visualisation fully integrating configuration management data with the source code will allow the difficulties of particular modifications to be monitored, by examining the time taken to make the change, and the structural effect on the rest of the project. Although animation has been used successfully within this research to show evolution, further research is also needed as to whether evolution may be illustrated more clearly using 2D or traditional 3D representations.

In this research, the use of animation alongside a 2D representation has created three dimensions in which to display data. The final extension that may be considered is to investigate the use of four-

dimensional visualisations – that is, 3D with time. As with 2D visualisations moving to 3D, the introduction of 3D with time may resolve some of the layout and scale issues involved with animated evolutionary visualisations. However, this would come at a cost of increased navigational difficulties and increased occlusion meaning some of the animation may go unnoticed. It is not possible to state whether the benefits of a 4D visualisation would outweigh the drawbacks without further research in this area.

4. Change blindness

Considering the length of time that human vision has been studied, change blindness is a very recent discovery. As described before, this refers to the fact that the visual memory possessed by humans is much smaller to that assumed, and therefore significant changes may occur to an image that go totally unnoticed by a user. The phenomenon has been shown to occur with very gradual changes, such as colour changes over a period of 30 seconds, or significant changes to an image, providing that there is a blank or a flashing effect between the two images.

Change blindness strikes hard at the assumption that the user will be aware of everything within a visualisation. Furthermore, it is not possible to assume that any changes made will be apparent – whether due to data changing dynamically, or through the use of grouping and filter operations. The effect of this may be that representations will need to consider the effects of data introduced and removed from a display, perhaps during a zoom or filter operation, and ensure that the user is completely aware of the changes made to the display. For example, the change circle used within HfVis exists to avoid problems of change blindness.

However, the property may also be beneficial within visualisation. With care, additional information could be added to the visualisation in a subconscious way, that would not significantly distract the user. For example, zooming in to an element within the visualisation may present further details about the element, such as the name, in an attached note. Normally, as the image was continuous, the user would be alerted to the arrival of the attached note, and others that were also displayed, and so become distracted from the element they were zooming into. Change blindness research suggests that if instead the note only appeared when the user was blinking, that they would be less distracted by the arrival of the additional note. Whether this is true in practice, and in what circumstances it may apply, is not clear. However, the benefit to a visualisation of being able to supplement the display unobtrusively means that this phenomenon warrants further research.

8.5. Conclusions

This research has shown the benefit of visualising software over a number of releases in order to both observe and understand the changes made within the software as well as to view the evolution of the project over time. In addition, the viability of using animation within software visualisation in order to create large scale, automatically generated visualisations has been demonstrated. Furthermore, this research as a whole has examined many of the issues that must be considered when visualising evolving software, and therefore provides a useful basis for further work in this area.

ANIMATING THE EVOLUTION OF SOFTWARE

References

[Ahlberg94] C. Ahlberg, B. Shneiderman, "Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays", *Proc. ACM Conf. Human Factors in Computing Systems*, New York, USA, 1994, pp. 313-317.

[Allegro03] *The Allegro graphics library*. http://alleg.sourceforge.net (June 2003)

[Allen90] R.E. Allen, "The concise Oxford dictionary of current English", 8th Edition, Oxford, UK, 1990.

[Andrews95] K. Andrews, "Visualising cyberspace: Information visualisation in the Harmony Internet browser", *Proc. Information Visualization*, Los Alamitos, USA, 1995, pp. 97-104.

[Ankerst96] M. Ankerst, D.A. Keim, H.-P. Kriegel, "'Circle Segments': A Technique for Visually Exploring Large Multidimensional Data Sets", *Proc. Visualization '96*, Hot Topic Session, San Francisco, USA, 1996

[Asklund01] U. Asklund, L. Bendix, "Configuration Management for Open Source Software", *Proc. 1st Workshop on Open Source Software Engineering,* Toronto, Canada, 2001.

[MBaker95] M.J. Baker, S.G. Eick, "Space-Filling Software Visualization", *J. Visual Languages and Computing,* vol. 6, no. 2, June 1995, pp. 119-133.

[BBaker95] B.S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems", *Proc. 2nd Working Conf. Reverse Engineering,* Toronto, Canada, 1995, pp. 86-95.

[Ball96] T.A. Ball, S.G. Eick, "Software Visualization in the Large", *IEEE Computer*, vol. 29, no. 4, Apr. 1996, pp. 33-43.

[Ball97] T.A. Ball et al., "If your version control system could talk", *Proc. Workshop Process Modelling and Empirical Studies of Software Engineering*, Boston, USA, 1997.

[Bartram01] L. Bartram, *Enhancing Information Visualization with Motion*, PhD Thesis, School of Computing Science, Simon Fraser Univ., Canada, 2001.

[Bartram01b] L. Bartram, C. Ware, T. Calvert, "Moving Icons: Detection and Distraction", *Proc. Interact 2001 Conference*, Waseda University, Japan, 2001.

[Bartram97] L. Bartram, *Perceptual and Interpretative Properties of Motion for Information Visualization*, tech. report CMPT-TR-1997-15, School of Computing Science, Simon Fraser Univ., Canada, 1997.

[Battista99] G.D. Battista et al., *Graph Drawing: Algorithms for the Visualization of Graphs,* Prentice Hall, USA, 1999.

[Baxter98] I.D. Baxter et al., "Clone Detection Using Abstract Syntax Trees", *Int'l Conf. Software Maintenance*, Bethesda, USA, 1998, pp. 368-377.

[Bederson98] B.B. Bederson, J. Meyer, "Implementing a Zooming User Interface: Experience Building Pad++", *J. Software: Practice and Experience,* vol. 28, no. 10, 1998, pp. 1101-1135.

[Bederson00] B.B. Bederson, J. Meyer, L. Good, "Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java", *Symp. User Interface Software and Technology,* San Diego, USA, 2000, pp. 171-180.

[Berliner90] B. Berliner, "CVS II: Parallelizing software development", *Proc. Winter 1990 USENIX Conference*, Washington D.C, USA, 1990, pp. 341-352.

[Binkley96] D. Binkley, K. Gallagher, "Program Slicing", *Advances in Computers volume 43*, M. Zelkowitz, ed., Academic Press, San Diego, USA, 1996, pp. 1-50.

[Bonsai03] *Bonsai.* http://bonzai.mozilla.org (Aug 2003)

[Brooks87] F.P. Brooks, "No silver bullet: essence and accidents of software engineering", *IEEE Computer* vol. 20, no. 4, Apr. 1987, pp. 10-19.

[Brown85] M. Brown, R.Sedgewick, "Techniques for algorithm animation", *IEEE Software,* vol. 2, no. 1, Jan. 1985, pp 28-39.

[Brown92] M. Brown, *Zeus: A system for algorithm animation and multi-view editing*, tech. report SRC-075, HP SRC Classic Lab, Palo Alto, USA, 1992.

[Brown93] M. Brown, M.A. Najork, *Algorithm Animation Using 3D Interactive Graphics*, tech. report SRC-110a, HP SRC Classic Lab, Palo Alto, USA, 1993.

[BugZilla03] *BugZilla.* http://www.bugzilla.org (Aug 2003)

[Burd02] E. Burd, D. Overy, A. Wheetman, "Evaluating Using Animation to Improve Understanding of Sequence Diagrams", *Int'l Workshop Program Comprehension*, Paris, France, 2002, pp. 107-113.

[Burd97] E. Burd, M. Munro, "Investigating the Maintenance Implications of the Replication of Code", *Int'l Conf. Software Maintenance*, Bari, Italy, 1997, pp. 322-330.

[Burd99] E. Burd, M. Munro, "An Initial Approach towards Measuring and Characterising Software Evolution", *Proc. 6th Working Conf. Reverse Engineering,* Los Alamitos, USA, 1999, pp. 168-174.

[Burkwald98] S.K. Burkwald et al., "Visualising Year 2000 Program Changes", *6th Int'l Workshop Program Comprehension*, Ischia, Italy, 1998, pp. 13-18.

[Byrne96] M.D. Byrne, R. Catrambone, J.T. Stasko, *Do Algorithm Animations Aid Learning?*, tech. report GIT-GVU-96-18, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, USA, 1996.

[Cain01] J. Cain, R. McCrindle, "Making Movies: Watching Software Evolve through Visualisation", *International Conference on Computation Science,* San Francisco, USA, 2001.

[Caprio01] F. Caprio, G. Casazza, M. Di Penta, U. Villano, "Measuring and Predicting the Linux Kernel Evolution", *Proc. Int'l Workshop Empirical Studies on Software Maintenance*, Florence, Italy, 2001.

[Card99] S.K. Card, J.D. McKinlay, B. Shneiderman eds., *"Readings in Information Visualisation - Using Vision to Think",* Morgan Kaufmann, San Francisco, USA, 1999.

[Carey99] R. Carey, G. Bell, *The online annotated VRML 97 reference*, http://www.web3d.org/resources/vrml_ref_manual/Book.html (June 2003)

[Carr99] D. Carr, "Guidelines for Designing Information Visualization Applications", *Proc. Ericsson Conference Usability Engineering*, Stockholm, Sweden, 1999.

[Chapin00] N. Chapin, "Software Maintenance Types - A Fresh View", *Int'l Conf. Software Maintenance,* San Jose, USA, 2000, pp. 247-253.

[Charters02] S.M.Charters et al. "Visualisation for Informed Decision Making; From Code to Components", *Proc. Workshop Software Engineering Decision Support, 14th Int'l Conf. Software Engineering and Knowledge Engineering*, Ischia, Italy, 15-19th July 2002.

[Chee97] C.L. Chee, S.S. Erdogan, "An installable Version Control File System for Unix", *Software Practice and Experience*, vol. 27, no. 6, June 1997, pp. 725-746.

[Chi98] E. Chi et al., "Visualizing the evolution of web ecologies", *Proc. ACM CHI 98 Conf. Human Factors in Computing Systems*, Los Angeles, USA, 1998, pp. 400-407.

[Cook90] Cook, C.R. and W. Harrison, "Insights on Improving the Maintenance Process Through Software Measurement," *Conference on Software Maintenance*, San Diego, USA, Nov. 1990, pp. 37-46.

[Cook00] S. Cook, H. Ji, R. Harrison, *Software Evolution and Software Evolvability*, working paper, Univ. Reading, UK, 2000.

[CsDiff03] *CsDiff*. http://www.componentsoftware.com/products/csdiff/index.htm

[Datrix03] *Datrix : Software System Analysis based on Source Code Reverse Engineering*. http://www.iro.umontreal.ca/labs/gelo/datrix/ (June 2003)

[Dijkstra68] E.W. Dijkstra, "Go To Statement Considered Harmful", *Comm. ACM*, vol. 11, no. 3, Mar. 1968, pp. 147-148.

[Doxygen03] *Doxygen*. http://www.doxygen.org/ (June 2003)

[Dutton99] R.T. Dutton, J.C. Foster, M.A. Jack, "Please mind the doors - do interface metaphors improve the usability of voice response services?", *BT Technology Journal*, vol. 17, no. 1, Jan. 1999, pp.172-177.

[Eick00] S.G. Eick, A.F. Karr, *Visual Scalability*, tech. report 106, National Institute of Statistical Sciences, North Carolina, USA, 2000.

[Eick01] S.G. Eick, et al., "Does Code Decay? Assessing the Evidence from Change Management Data", *IEEE Trans. Software Engineering*, vol. 27, no. 1, Jan. 2001, pp. 1-12.

[Eick02] S.G. Eick et al, "Visualizing Software Changes", *IEEE Trans. Software Engineering*, vol. 28, no. 4, Apr. 2002, pp. 396-412.

[Eick92] S.G. Eick et al., "SeeSoft - A Tool for Visualizing Line Oriented Software Statistics", *IEEE Trans. Software Engineering*, vol. 18, no. 11, Nov. 1992, pp. 957-968.

[Estublier00] J. Estublier, "Software configuration management: a roadmap", *Int'l Conf. Software Engineering – Future of SE Track*, Limerick, Ireland, 2000, pp. 279-289.

[Fekete02] J.-D. Fekete, C. Plaisant, "Interactive Information Visualization of a Million Items", *Proc. IEEE Symp. Information Visualization 2002*, Boston, USA, 2002, pp. 117-124.

[Ferenc02] R. Ferenc et al, "Columbus - Reverse Engineering Tool and Schema for C++", *Int'l Conf. Software Maintenance*, Montreal, Canada, 2002, pp. 172-181.

[Flash03] *Macromedia Flash*. http://www.macromedia.com/software/flash/ (June 2003)

[FSF03] *Philosophy of the GNU - Free Software Foundation*. http://www.gnu.org/philosophy/philosophy.html (June 2003)

[Furnas81] G.W. Furnas, *The FISHEYE view: a new look at structured files*, tech. report 81-11221-9, Bell Laboratories, Murray Hill, USA, 1981.

[Furnas86] G.W. Furnas, "Generalized Fisheye Views", *Proc. ACM CHI '86 Human Factors in Computing Systems*, Boston, USA, 1986, pp.18-23.

[Gacek02] C. Gacek, T. Lawrie, B. Arief, "Interdisciplinary Insights on Open Source", *Proc. Open Source Software Development Workshop*, Newcastle, UK, 2002, pp. 68-82.

[Gall99] H. Gall, M. Jazayeri, C. Riva, "Visualizing Software Release Histories: The Use of Color and Third Dimension", *Proc. Int'l Conf. Software Maintenance*, Oxford, UK, 1999, pp. 99-108.

[Godfrey00] M.W. Godfrey, Q. Tu, "Evolution in Open Source Software: A Case Study", *Proc. Int'l Conf. Software Maintenance*, San Jose, USA, 2000, pp. 131-142.

[Graves00] T.L. Graves et al., "Predicting Fault Incidence Using Software Change History", *Trans. Software Engineering,* vol. 26, no. 7, July 2000, pp. 653-661.

[Graves98] T.L. Graves, A. Mockus, "Inferring change effort from configuration management data", *5th Int'l Symp. Software Metrics*, Bethesda, USA, 1998, pp. 267-273.

[Hatch01] A. Hatch, M. Smith, C. Taylor, M. Munro, "No Silver Bullet for Software Visualisation Evaluation", *Proc. Int'l Conf. Imaging Science, Systems, and Technology*, Las Vegas, USA, 2001, pp. 651-657.

[Hoek00] A. Hoek, "Configuration management and Open Source projects.", *Proc. 3rd Int'l Workshop Software Engineering over the Internet*, Limerick, Ireland, 2000.

[Horwitz90] S. Horwitz, "Identifying the Semantic and Textual Differences Between Two Versions of a Program", *Proc. ACM Conf. Programming Language Design and Implementation,* New York, USA, 1990, pp. 243-245.

[Hubb96] R.J. Hubbold, X. Dongbo, S. Gibson, "MAVERIK - the Manchester Virtual Environment Interface Kernel", *Proc. 3rd Eurographics Workshop on Virtual Environments*, Monte-Carlo, 1996.

[Humphrey88] W.S. Humphrey, "Characterizing the Software Process: A Maturity Framework", *IEEE Software*, vol. 5, no. 2, Mar. 1988, pp. 73-79.

[Hunt77] J.W. Hunt, T.G. Szymanski, "A Fast Algorithm for Computing Longest Common Subsequences", *Comm. ACM,* vol. 20, no. 5, May 1977, pp. 350-353.

[Hunt98] J.J. Hunt, K-P. Vo, W. Tichy, "Delta Algorithms: An Empirical Analysis", *ACM Trans. Software Engineering and Methodology*, vol. 7, no. 2, Apr. 1998, pp. 192-214.

[IEEE94] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990 (1991 Corrected Edition), The Institute of Electircal and Electronics Engineers, Inc., 1994.

[Java3D] *Java3D*. http://java.sun.com/products/java-media/3D/ (June 2003)

[JavaDoc03] *JavaDoc*. http://java.sun.com/j2se/javadoc/ (June 2003)

[Jog95] N. Jog, B. Shneiderman, "Starfield Information Visualization with Interactive Smooth Zooming", *Proc. Visual Databases Systems,* Lausanne, Switzerland, 1995, pp. 1-10.

[Johnson91] B. Johnson, B.Shneiderman, "Tree-maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures", *Proc. Visualization*, Los Alamitos, USA, 1991, pp. 284-291.

[Johnson94] J.H. Johnson, "Substring Matching for Clone Detection and Change Tracking", *Int'l Conf. Software Maintenance*, Victoria, Canada, 1994, pp. 120-126.

[Jones01] J.A. Jones, M.J. Harrold, J.T. Stasko, "Visualization for Fault Localization", *Proc. ICSE 2001 Workshop on Software Visualization*, Toronto, Canada, 2001, pp. 71-75.

[Jørgensen95] M.Jørgensen, "An Empirical Study of Software Maintenance Tasks", *J. Software Maintenance*, vol. 7, no. 1, Jan./Feb. 1995, pp. 27-48.

[Kehoe99] C. Kehoe, J. Stasko, A. Taylor, *"Rethinking the Evaluation of Algorithm Animations as Learning Aids: An Observational Study"*, tech. report GIT-GVU-99-10, Graphics, Visualization, and Usability Center, College of Computing, Georgia Institute of Technology, Atlanta, USA, 1999.

[Keim94] D.A. Keim, H.-P. Kriegel, "VisDB: Database Exploration using Multidimensional Visualisation", *IEEE Computer Graphics and Applications,* vol. 14, no. 5, Sept./Oct. 1994, pp. 40-49.

[Keim95] D.A. Keim, H.-P. Kriegel, M. Ankerst, "Recursive Pattern: A Technique for Visualizing Very Large Amounts of Data", *Proc. IEEE Visualization '95*, Atlanta, USA, 1995, pp. 279-287.

[Keim96] D.A. Keim, H.-P. Kriegel, "Visualization Techniques for Mining Large Databases: A Comparison", *IEEE Trans. Knowledge and Data Engineering*, vol. 8, no. 6, Dec. 1996, pp. 923-938.

[Kenwood02] C. Kenwood, "A Business Case Study of Open Source Software", *Proc. Open Source Software Development Workshop*, Newcastle, UK, 2002.

[Kitchenham96] B. Kitchenham, L. Jones, "Evaluating Software Engineering Methods and Tools. Part 1: The Evaluation Context and Evaluation Methods", *Software Engineering Notes*, vol. 21, no. 1, Jan. 1996, pp. 12-15.

[Knight00a] Virtual Software in Reality, PhD Thesis, Dept. Computer Science, Univ. of Durham, UK, June 2000.

[Knight00b] C. Knight, M.Munro, "Mindless Visualisations", *Proc. 6th ERCIM User Interfaces for All Workshop*, Florence, Italy, 2000.

[Knight99] C. Knight, M. Munro, "Comprehension with[in] Virtual Environment Visualisations", *7th Int'l Workshop on Program Comprehension*, Pittsburgh, USA, 1999, pp. 4-11.

[Koch00] S. Koch, G. Schneider, *Results from Software Engineering Research into Open Source Development Projects Using Public Data*, tech. report, Dept. Information Business, Univ. of Vienna, Austria. 2000.

[Koike93] H. Koike, "The Role of Another Spatial Dimension in Software Visualization", *ACM Trans. Information Systems*, vol. 11, no. 3, July 1993, pp. 266-286.

[Koike97] H. Koike, H.-C. Chu, "VRCS: Integrating Version Control and Module Management using Three-Dimensional Visualization", *Proc. 1997 IEEE Symposium on Visual Languages*, Capri, Italy, 1997, pp. 170-175.

[Korn02] D. Korn, K.-P. Vo, "Engineering a Differencing and Compression Data Format", *USENIX Annual Technical Conference,* Monterey, USA, 2002, pp. 219-228.

[LaFolette00] P. LaFolette, J. Korsh, R. Sangwan, "A Visual Interface for Effortless Animation of C/C++ Programs", *J. Visual Languages and Computing,* vol. 11, no. 1, Feb. 2000, pp. 27-48.

[Lahtinen98] S.P. Lahtinen, E. Sutinen, J. Tarhio, "Automated Animation of Algorithms with Eliot", *J. Visual Languages and Computing,* vol. 9, no. 3, June 1998, pp. 337-349.

[Lamping96] J. Lamping, R. Rao, "The Hyperbolic Browser: A Focus + Context Technique for Visualizing Large Hierarchies", *J. Visual Languages and Computing*, vol. 7, no. 1, March 1996, pp. 33-35.

[Lanza01] M. Lanza, "The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques", *Proc. Int'l Workshop Principles of Software Evolution*, Vienna, Austria, 2001, pp. 37-42.

[Larkin87] J.H. Larkin, H.A. Simon, "Why a diagram is (sometimes) worth 10,000 words", *Cognitive Science*, vol. 11, no. 1, Jan. 1987, pp. 65-99.

[Lawrence94] A.W. Lawrence, A.N. Badre, J.T. Stasko, "Empirically Evaluating the Use of Animations to Teach Algorithms". *Proc. 1994. IEEE Symp. Visual Languages,* St. Louis, USA, 1994, pp. 48-54.

[Leblang94] D.B. Leblang, "The CM Challenge: Configuration Management that Works", *Configuration Management*, W.F. Tichy, ed., Wiley and Sons Ltd, New York, USA, 1994.

[Lehman00] M.M. Lehman, *Rules and Tools for Software Evolution Planning and Management*, tech. report 2000/14, Dept. Computer Science, Imperial College, London, UK, 2000.

[Lehman01] M.M. Lehman, *FEAST/2 Final Report - Grant Number GR/M44101*, tech. report, Dept. of Computer Science, Imperial College, London, UK, 2001.

[Lehman85] M.M. Lehman, L.A. Belady, eds., *Program Evolution : Processes of Software Change*, Academic Press, London, UK, 1985.

[MacDonald98] J. MacDonald, P.N. Hilfinger, L. Semenzato, "PRCS: The Project Revision Control System", *Proc. 8th Int'l Symp. System Configuration Management*, Brussels, Belgium, 1998, pp. 33-45.

[Mackinlay91] J.D. Mackinlay, S.K. Card, G.C. Robertson, "Perspective Wall: Detail and Context Smoothly Integrated", *Proc. Human Factors in Computing Systems*, New Orleans, 1991, pp. 173-179.

[Make03] *Make*. http://www.gnu.org/software/make/ (June 2003)

[Maletic02] J.L. Maletic, A. Marcus, M. Collard, "A Task Oriented View of Software Visualization", *Proc. IEEE Workshop on Visualizing Software for Understanding and Analysis*, Paris, France, 2002, pp. 32-40.

[Mattsson00] M. K-Mattsson, "Preventative Maintenance! Do we know what it is?", *Int'l Conf. Software Maintenance,* San Jose, USA, 2000, pp. 12-13.

[Mayrand96] J. Mayrand, C. Leblanc, E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", *Proc. Int'l Conf. Software Maintenance,* Monterey, USA, 1996, pp. 244-253.

[Mayrhauser95] A. von Mayrhauser, A.M. Vans, "Program Comprehension During Software Maintenance and Evolution", *IEEE Computer*, vol. 28, no.8, August 1995, pp. 44-55.

[Miller85] W. Miller, E.W. Myers, "A File Comparison Program", *Software - Practice and Experience*, vol. 15, no. 11, Nov. 1985, pp. 1025-1040.

[Ming03] *Ming - an SWF output library and PHP module.* http://ming.sourceforge.net/ (July 03)

[Mockus00a] A. Mockus, L.G. Votta, "Identifying Reasons for Software Changes using Historic Databases", *Proc. Int'l Conf. Software Maintenance,* San Jose, USA, 2000, pp. 120-130.

[Mockus00b] A. Mockus, D.M. Weiss, "Predicting risk of software changes", *Bell Labs Technical Journal*, vol. 5, no. 2, Apr.-Jun. 2000, pp. 169-180.

[Mockus00c] A. Mockus, R.T. Fielding, J.D. Herbsleb, "A case study of open source software development: the Apache server", *Proc. 22nd Int'l Conf. Software Engineering,* Limerick, Ireland, 2000, pp. 263-272.

[Mockus02] A. Mockus, R.T. Fielding, J.D. Herbsleb, "Two case studies of open source software development: Apache and Mozilla", *ACM Trans. Software Engineering and Methodology*, vol. 11, no. 3, 2002, pp. 309-346.

[Myers90] B.A. Myers, "Taxonomies of Visual Programming and Program Visualization", *J. Visual Languages and Computing*, vol. 1, no. 1, Mar. 1990, pp. 97-123.

[North96] S. North, "Incremental layout in DynaDAG", *Graph Drawing, vol. 1027 Lecture Notes Computer Science*, F. J. Brandenburg, ed., Springer-Verlag, 1996, pp. 409-418.

[Ohlsson99] M. Ohlsson et al., "Code Decay Analysis of Legacy Software through Successive Releases", *Proc. IEEE Aerospace Conf*, Colorado, USA, 1999.

[OSI03] *Open Source Initiative.* http://www.opensource.org/ (June 2003)

[Petre96] M. Petre, A.F. Blackwell, T.R.G. Green, "Cognitive Questions in Software Visualisation", *Software Visualisation: Programming as a Multi-Media Experience,* J. Stasko, J. Domingue, M. Brown & B. Price, eds., MIT Press, Cambridge, USA, 1998, pp. 453-480.

[Pigoski97] T.M. Pigoski, *Practical software maintenance: best practices for managing your software investment,* John Wiley & Sons Inc, New York, USA, 1997.

[Price93] B.A. Price, R.M. Baecker, I.S. Small, "A Principled Taxonomy of Software Visualization", *J. Visual Languages and Computing*, vol. 4, no. 3, Sept. 1993, pp. 211-266.

[Purchase02] H.C. Purchase, D. Carrington, J.-A. Allder, "Empirical Evaluation of Aesthetics-based Graph Layout", *Empirical Software Engineering*, vol. 7, no. 3, 2002, pp. 233-255.

[Quigley02] A. Quigley, "Experience with FADE for the Visualization and Abstraction of Software Views", *Proc. Int'l Workshop Program Comprehension,* Paris, France, 2002, pp. 11-20.

[Rajlich00] V. Rajlich, K.H. Bennett, "A Staged Model for the Software Life Cycle", *IEEE Computer*, vol. 33, no. 7, July 2000, pp. 66-71.

[Ramil00] J.F. Ramil, M.M. Lehman, "Cost Estimation and 'Evolvability' Monitoring for Software Evolution Processes", *Workshop Empirical Studies of Software Maintenance*, San Jose, USA, 2000.

[Rao94] R. Rao, S.K. Card, "The Table Lens: Merging Graphical and Symbolic Representations in an Interactive Focus+Context Visualization for Tabular Information", *Proc. ACM Conf. Human Factors in Computing Systems*, Boston, USA, 1994, pp. 318-322.

[Raymond99] E.S. Raymond, *The cathedral and the bazaar : musing on Linux and Open Source by an accidental revolutionary*, O'Reilly, Cambridge, USA, 1999.

[Reis02] C.R. Reis, R.P.M. Fortes, "An overview of the Software Engineering Process in the Mozilla project", *Proc. Open Source Software Development Workshop*, Newcastle, UK, 2002, pp. 155-175.

[Rekimoto93] J. Rekimoto, M. Green, "The Information Cube: Using Transparency in 3D Information Visualisation", *Proc. 3rd Annual Workshop on Information Technologies & Systems*, Orlando, USA, 1993, pp. 125-132.

[Rensink02] R.A. Rensink, "Change Detection", *Annual Review of Psychology*, vol. 53, 2002, pp. 245-277.

[Riva98] C. Riva, *Visualizing Software Release Histories: The Use of Color and Third Dimension*, MSc Thesis, Dept. Computer Science, Univ. of Vienna, Austria, 1998.

[Robertson91] G.G. Robertson, J.D. Mackinlay, S.K. Card, "Cone Trees: animated 3D visualizations of hierarchical information", *Proc. Human Factors in Computing Systems*, New Orleans, USA, 1991, pp. 189-194.

[Robertson93] G.G. Robertson, J.D. Mackinlay, "The Document Lens", *Proc. ACM Symposium on User Interface Software and Technology*, Atlanta, Georgia, USA, 1993, pp. 101-108.

[Roman92] G.C. Roman et al., "Pavane: a System for declarative Visualization of Concurrent Computations", *J. Visual Languages and Computing,* vol. 3, no. 2, June 1992, pp. 161-193.

[Roman93] G.C. Roman, K.C. Cox, "A Taxonomy of Program Visualization Systems", *IEEE Computer,* vol. 26, no. 12, Dec. 1993, pp. 11-23.

[Santos00] C. Russo Dos Santos et al., "Experiments in Information Visualization using 3D Metaphoric Worlds", *IEEE 9th Int'l Workshop Enabling Technologies: Infrastructure for Collaborative Enterprises*, Gaithersburg, USA, 2000, pp. 51-58.

[Sarkar94] M. Sarkar, M. Brown, "Graphical Fisheye Views", *Comm. ACM*, vol. 37, no. 12, Dec. 1994, pp. 73-84.

[Scaife96] M. Scaife, Y. Rogers, "External cognition: how do graphical representations work?", *Int'l J. Human-Computer Studies*, vol. 45, no.2 ,August 1996, pp. 185-213.

[Schneidewind87] N.F. Schneidewind, "The State of Software Maintenance", *IEEE Trans. Software Engineering*, vol. SE-13, no. 3, Mar. 1987, pp. 303-310.

[Scott00] K.C. Scott-Brown, M.R. Baker, H.S. Orbach, "Comparison blindness". *Visual Cognition*, vol. 7, no. 1-3, 2000, pp. 254-267.

[Shneiderman96] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations", *Proc. IEEE Visual Languages*, Colorado, USA, 1996, pp. 336-343.

[Simons00] D.J. Simons, S.L. Franconeri, R.L. Reimer, "Change blindness in the absence of visual disruption", *Perception*, vol. 29, no.10, October 2000, pp. 1143-1154.

[Smith02] M.P. Smith, M. Munro, "Runtime Visualisation of Object Oriented Software", *Proc. Workshop Visualizing Software for Understanding and Analysis*, Paris, France, 2002, pp. 81-89.

[So02] H. So, N. Thomas, H. Zadeh, "What is in a Bazaar? A Model of Individual Participation in an Open Source Community", *Proc. Open Source Software Development Workshop*, Newcastle, UK, 2002, pp. 101-121.

[Stasko00] *GVU Homepage*. http://www.cc.gatech.edu/gvu/softviz/ (June 2003)

[Stasko90] J. Stasko, "TANGO: A Framework and System for Algorithm Animation", *IEEE Computer*, vol. 23, no. 9, Sept. 1990, pp. 27-39.

[Stasko92] J.T. Stasko, E. Kraemer, *A Methodology for Building Application-Specific Visualizations of Parallel Programs*, tech. report GIT-GVU-92-10, Graphics, Visualization and Usability Center, College of Computing, Georgia Institute of Technology, Atlanta, USA, 1992.

[Stasko96] J. Stasko, *The Information Mural: Increasing Information Bandwidth in Visualizations*, tech. report GIT-GVU-96-25, Graphics, Visualization, and Usability Center, College of Computing, Georgia Institute of Technology, Atlanta, USA, 1996.

[Storey97] M.A.D. Storey, F.D. Fracchia, H.A. Müller, "Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration", *Proc. 5th Int'l Workshop Program Comprehension*, Dearborn, USA, 1997, pp. 17-28.

[Storey01] M.-A.D. Storey, C. Best, J. Michaud, "ShriMP Views: : An Interactive and Customizable Environment for Software Exploration", *Int'l. Workshop Program Comprehension,* 2001, Toronto, Canada, pp. 111-112.

[Tichy85] W.F. Tichy, "RCS – A system for version control", *Software – Practice and Experience,* vol. 15, no. 7, July 1985, pp. 637-654.

[Tichy03] W.F. Tichy, *RCE vs RCS*, http://wwwipd.ira.uka.de/~RCE/ (June 2003)

[Tu02] Q. Tu, M.W. Godfrey, "An integrated approach for studying software architectural evolution", *Proc. Int'l Workshop Programming Comprehension*, Paris, France, 2002, pp. 127-136.

[Tufte83] E.R. Tufte, *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, Connecticut, USA, 1983.

[ViewCVS03] *ViewCVS*. http://viewcvs.sourceforge.net/ (June 2003)

[Vince00] J. Vince, "Essential Computer Animation", Springer-Verlag, London, UK, 2000.

[Ware94] C. Ware, S. Limoges. *Perceiving Data Displayed Through Oscillatory Motion.* tech. report TR94-089, Dept. Computer Science, Univ. of New Brunswick, Canada. 1994.

[Weber01] M. Weber, M. Alexa, W. Müller, "Visualizing Time-Series on Spirals", *IEEE Symp. Information Visualization 2001*, San Diego, California, USA, 2001, pp. 7-14.

[Wills99] G.J. Wills, "NicheWorks - Interactive Visualization of Very Large Graphs", *J. Computational and Graphical Statistics*, vol. 8, no. 2, 1999, pp. 190-212.

[WindRiver03] *Sniff+*. http://www.windriver.com/products/sniff_plus/index.html (Aug 2003)

[Yamauchi00] Y. Yamauchi et al., "Collaboration with Lean Media: How Open-Source Software Succeeds", *Proc. ACM Conf. Computer Supported Cooperative Work*, Philadelphia, USA, 2000, pp. 329-338.

[Yang89] W. Yang, S. Horwitz, T. Reps, *Detecting program components with equivalent behaviors*, tech. report TR 840, Computer Sciences Dept., Univ. of Wisconsin, USA, 1989.

[Yang91] W. Yang, "Identifying Syntactic Differences Between Two Programs", *Software – Practice and Experience,* vol. 21, no. 7, July 1991, pp. 739-755.

[Yee01] K.-P. Yee et al., "Animated Exploration of Graphs with Radial Layout", *Proc. IEEE Symp. Information Visualization*, San Diego, USA, 2001, pp. 43-50.

[Young99] P. Young, *Visualising Software in Cyberspace*, PhD Thesis, Dept. Computer Science, Univ. of Durham, UK, 1999.